LECTURE 5 : Sorting II

SEHH2239 Data Structures

Learning Objectives:

- To able to use the divide and conquer
- To describe and implement Merge sort and Quick sort

Divide and Conquer

Divide and Conquer

Divide-and-conquer algorithms generally have best complexity when a large instance is divided into smaller instances of approximately the same size.

1. Base Case, solve the problem directly if it is small enough
2. Divide the problem into two or more similar and smaller subproblems
3. Recursively solve the subproblems
4. Combine solutions to the subproblems

Divide and Conquer - Sort

Problem:

- Input: A[left..right] – **unsorted** array of integers
- Output: A[left..right] – **sorted** in non-decreasing order

Examples are Merge Sort and Quick Sort

Divide and Conquer - Sort

1. Base case

   at most one element (left ≥ right), return

2. Divide A into two subarrays: FirstPart, SecondPart

   Two Subproblems:

   sort the FirstPart

   sort the SecondPart

3. Recursively

   sort FirstPart

   sort SecondPart

4. Combine sorted FirstPart and sorted SecondPart

Merge Sort

Merge sort

- Merge sort keeps on dividing the list into equal halves until it can no more be divided.
- By definition, if it is only one element in the list, it is sorted.
- Then, merge sort combines the smaller sorted lists keeping the new list sorted too.

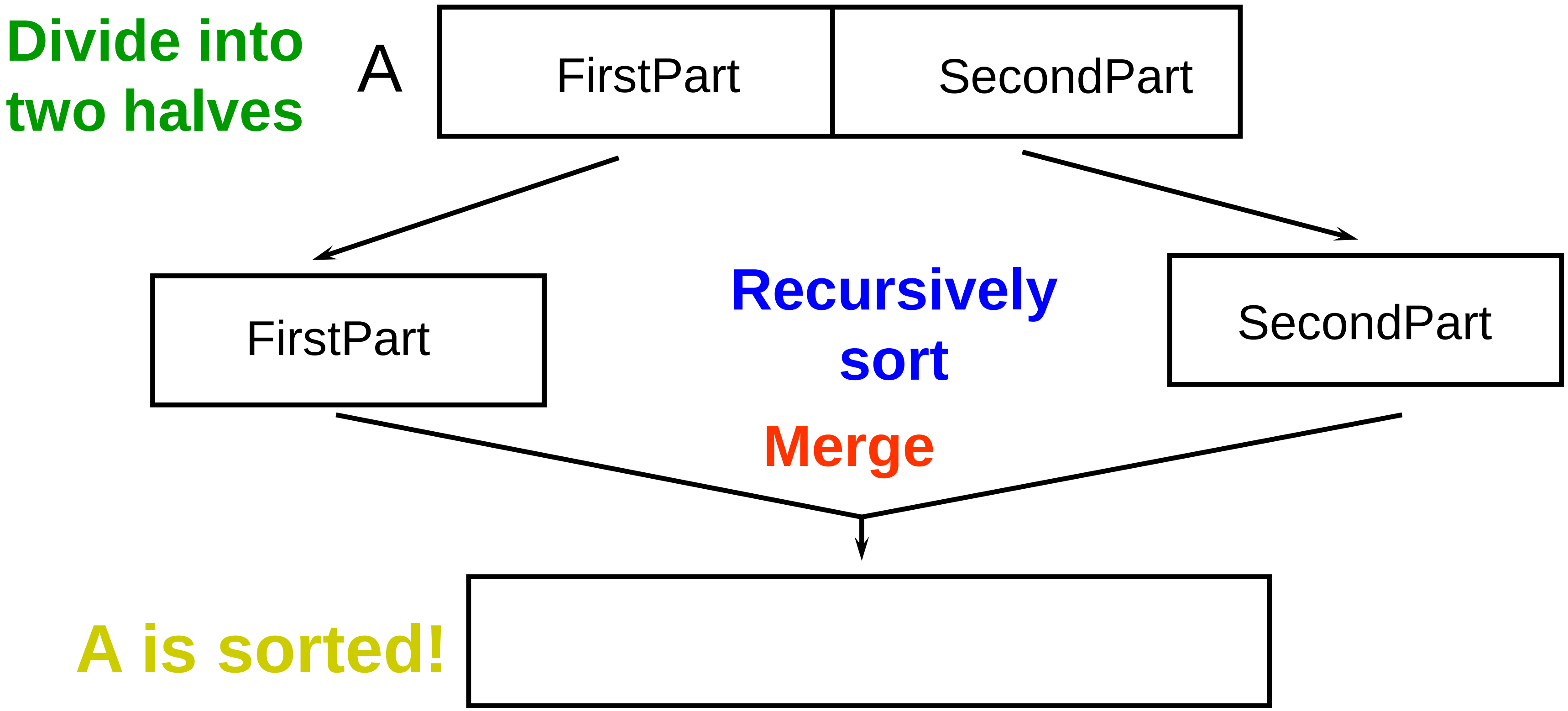
Merge Sort

- First ceil(n/2) elements define one of the smaller instances; remaining elements define the second smaller instance.
- Each of the two smaller instances is sorted recursively.
- The sorted smaller instances are combined using a process called **merge**.

ceil(n/2) $\boxed{\left\lceil \frac{25}{2} \right\rceil = 13}$

floor(n/2) $\boxed{\left\lfloor \frac{25}{2} \right\rfloor = 12}$

Merge Sort: Idea



Merge Sort: Algorithm

Merge-Sort (A, left, right)

if left $\geq$ right return

else

middle $\leftarrow \lfloor$(left+right)/2$\rfloor$

Merge-Sort(A, left, middle)

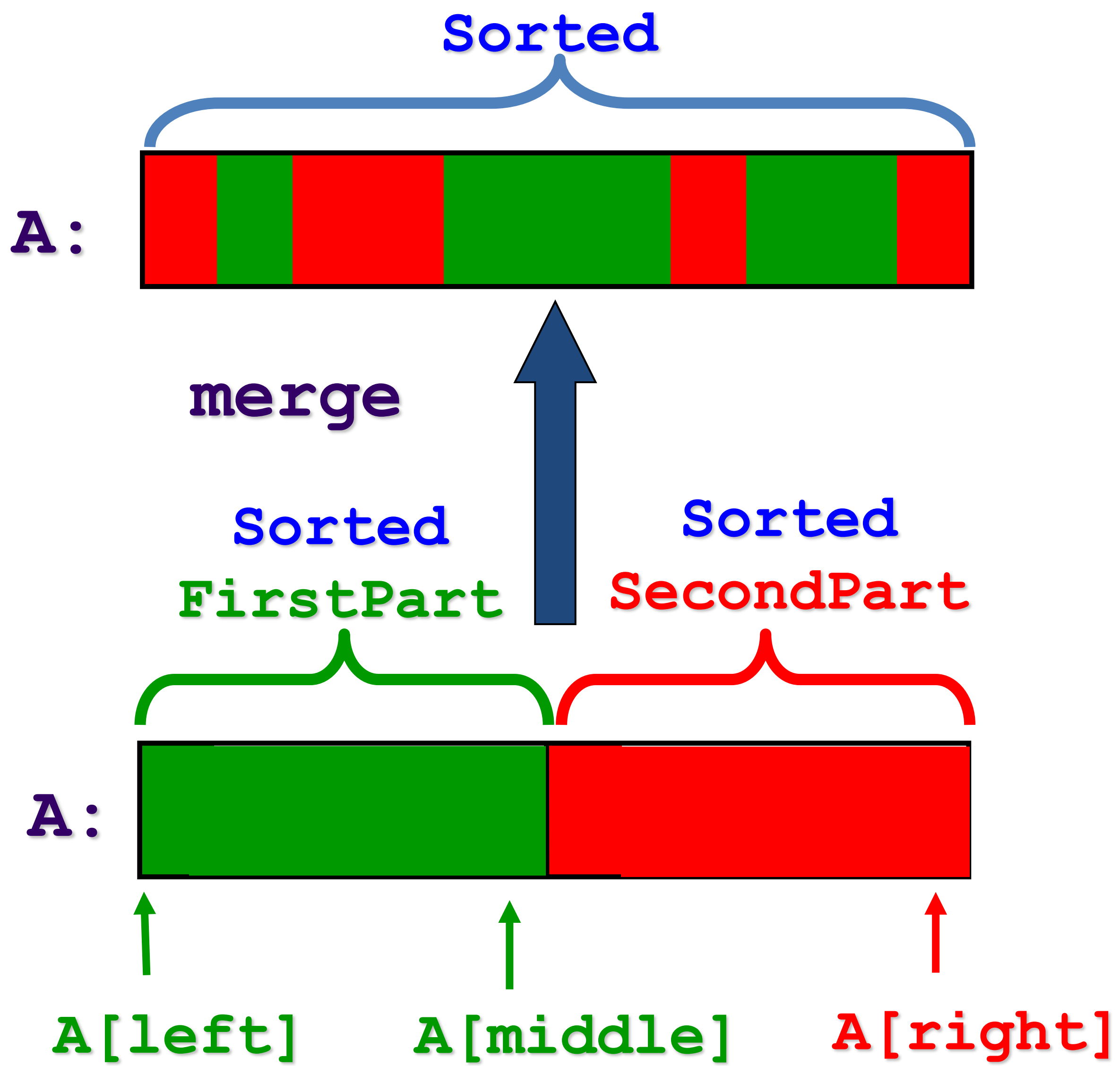
Recursive Call

Merge-Sort(A, middle+1, right)

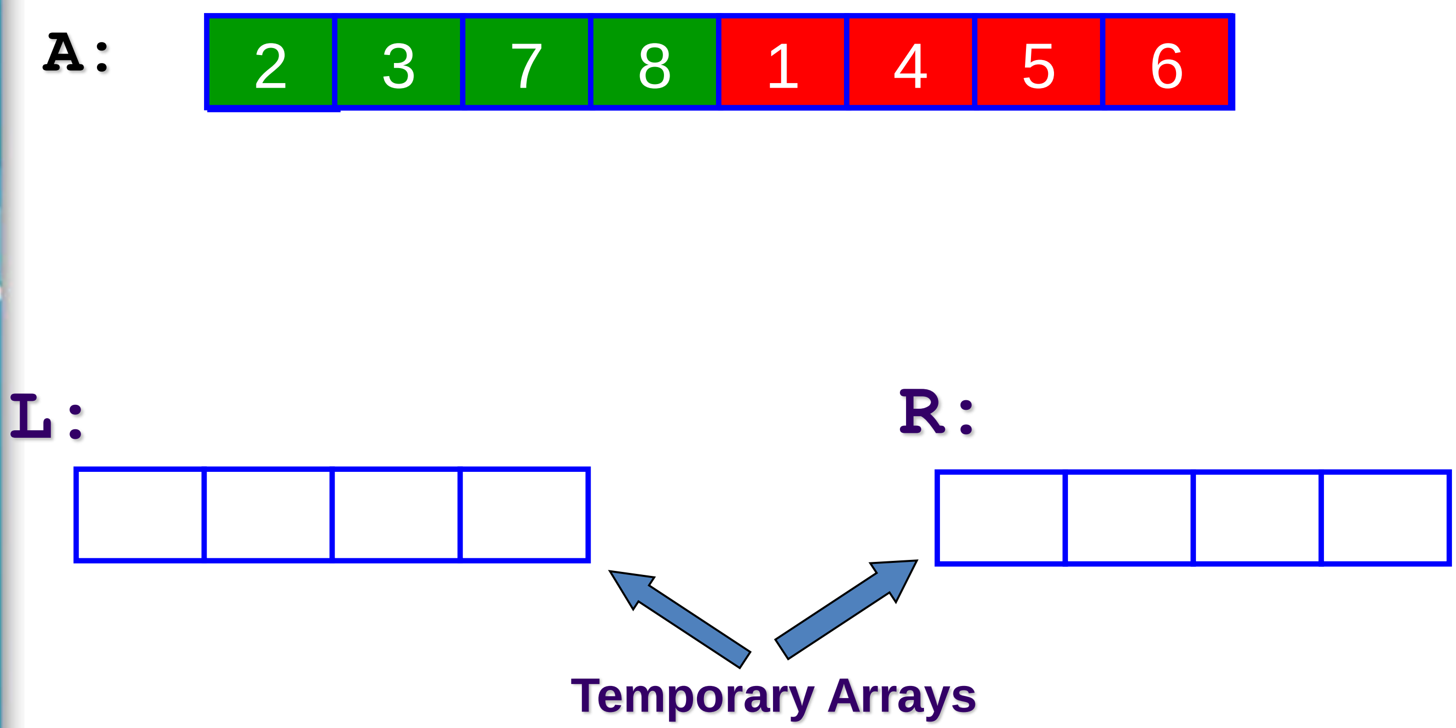
Merge(A, left, middle, right)

Merging with Array

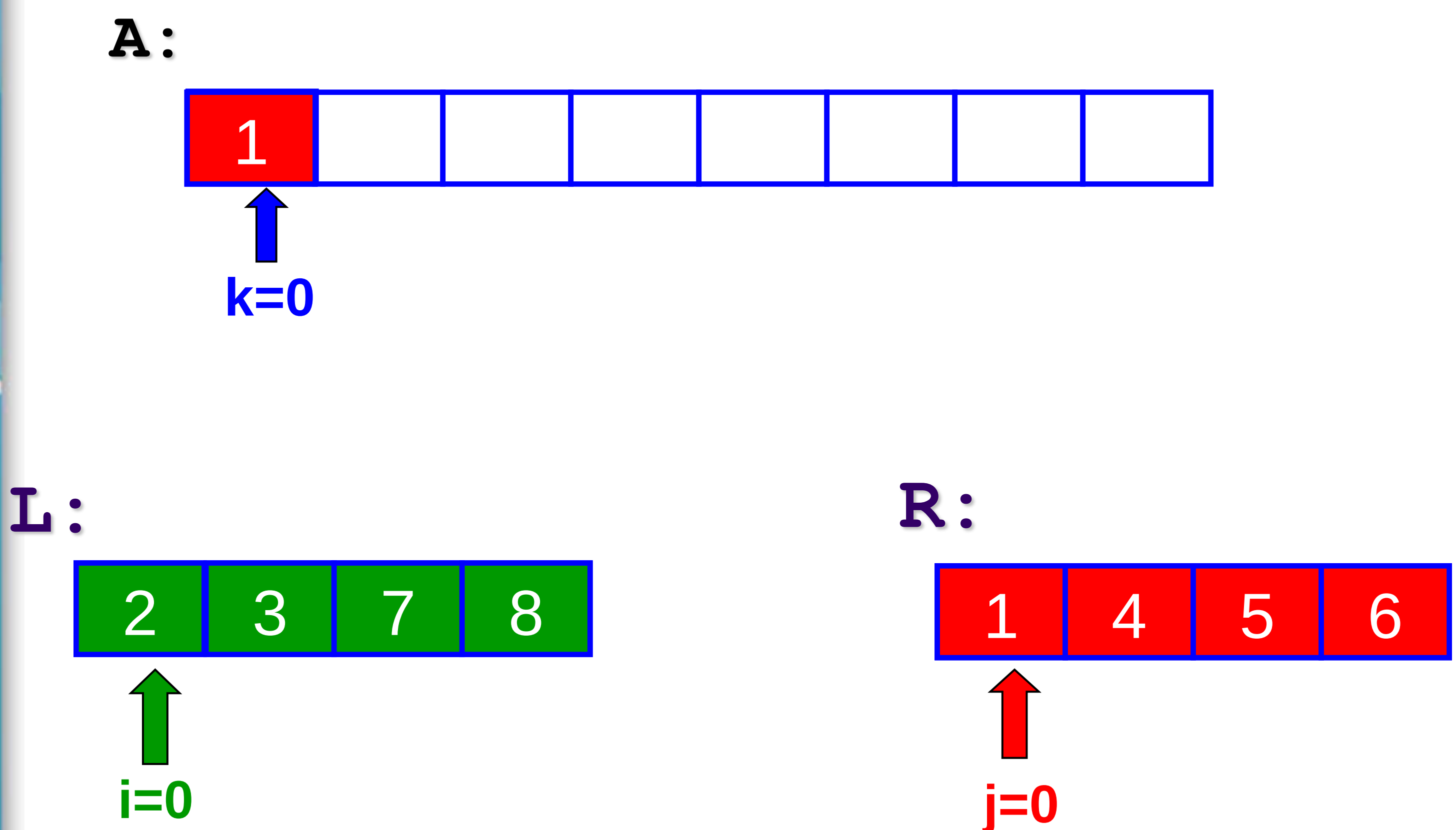
Merge-Sort: Merge



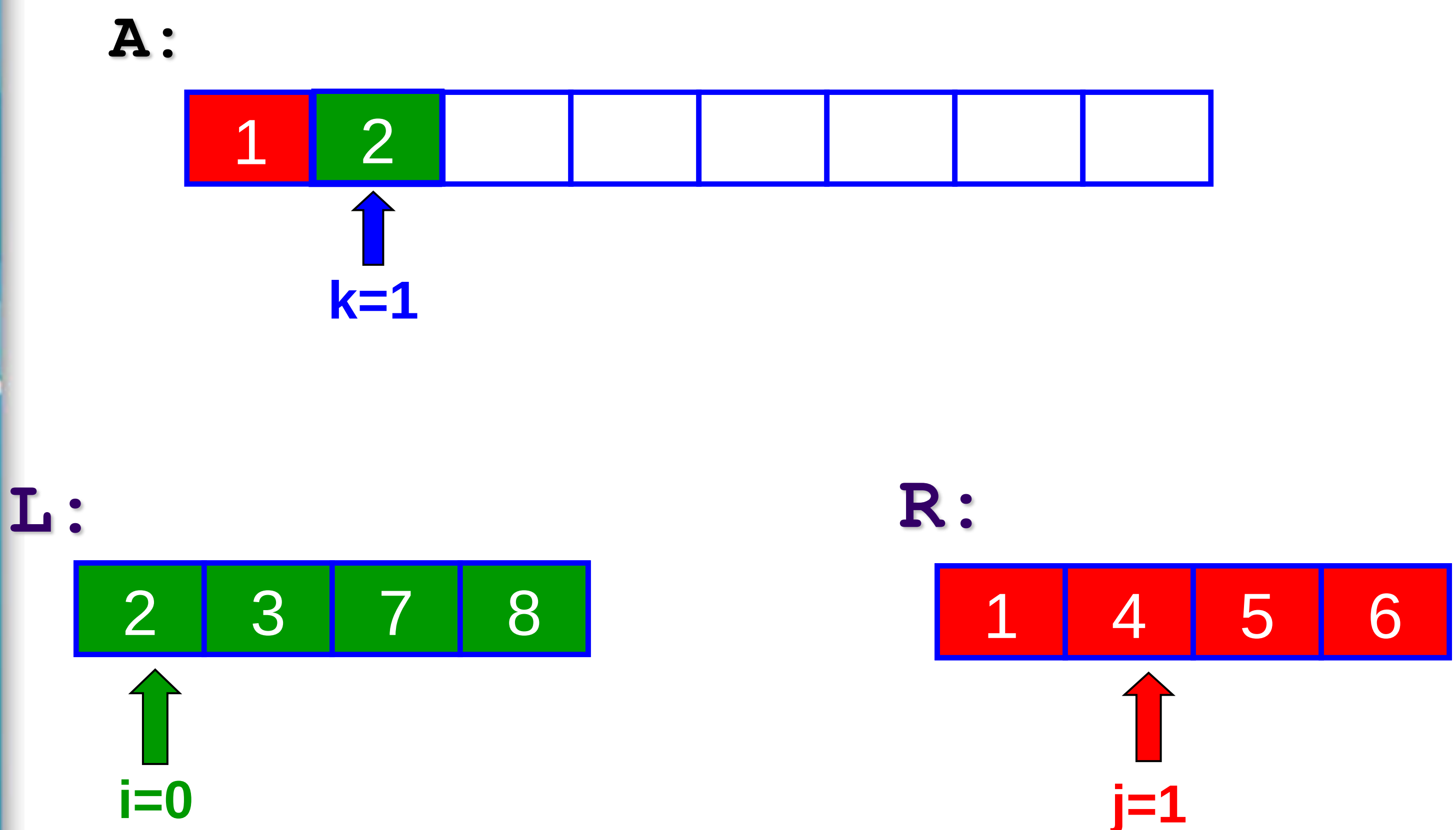
Merge-Sort: Merge Example



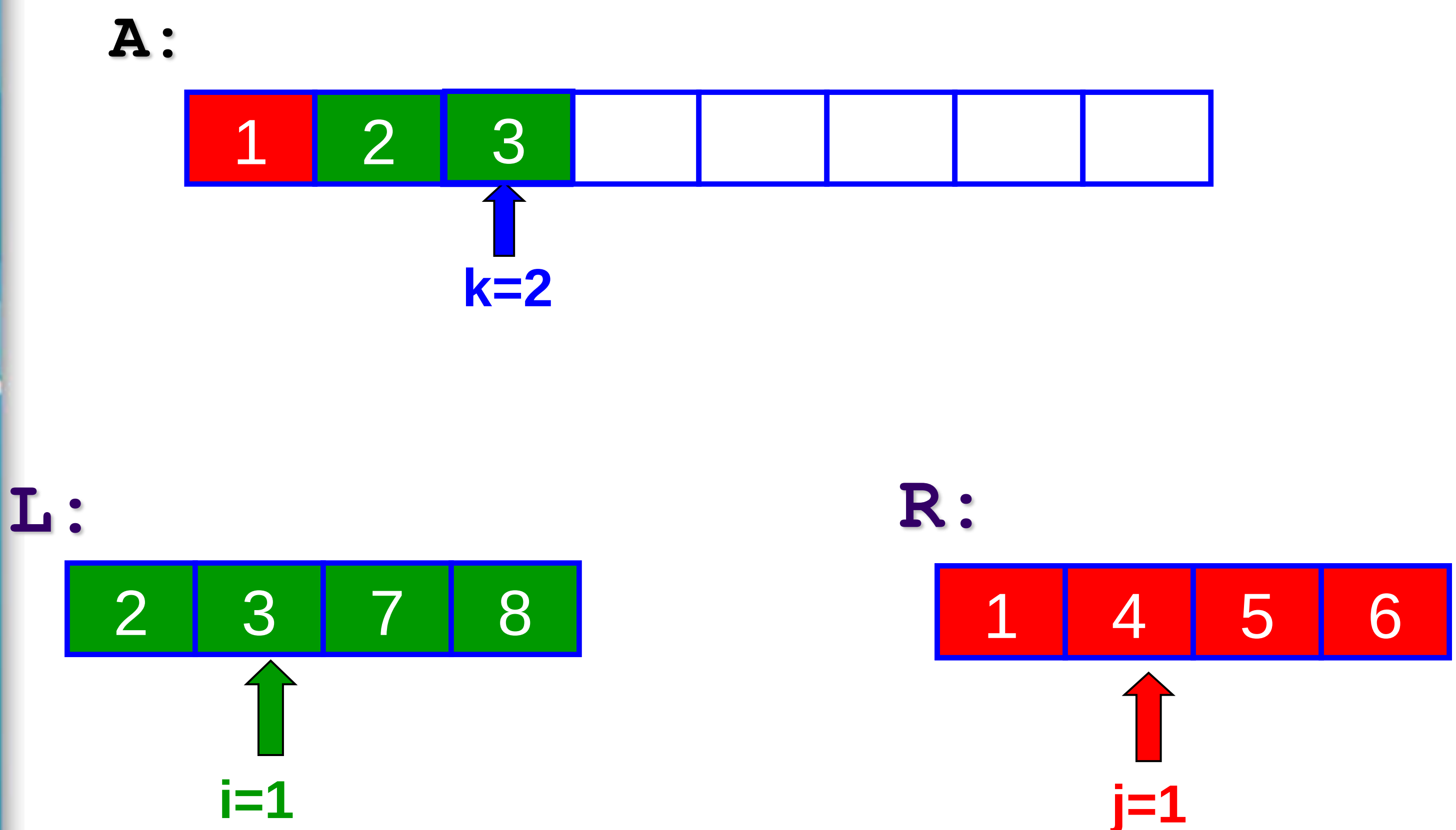
Merge-Sort: Merge Example



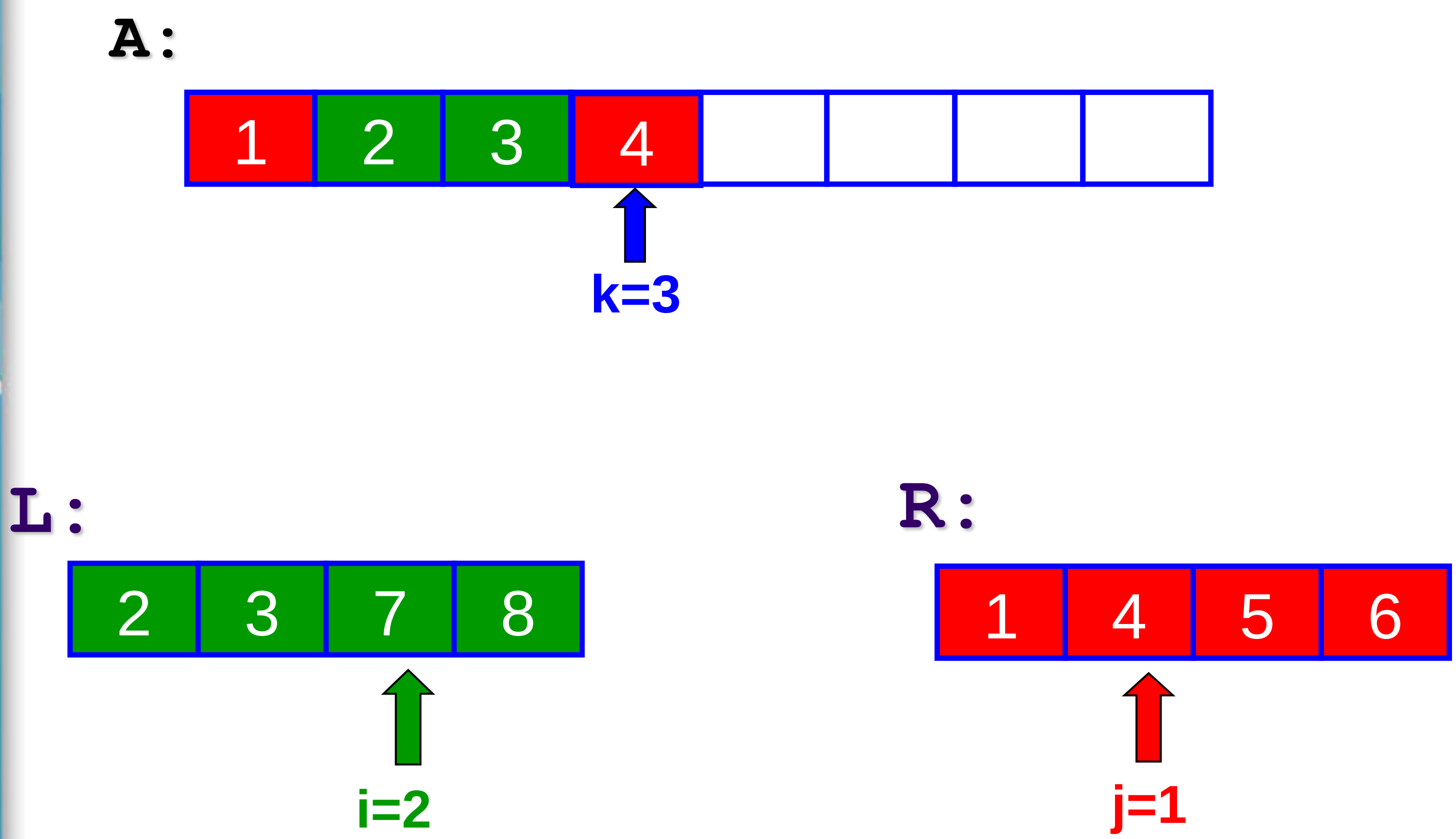
Merge-Sort: Merge Example



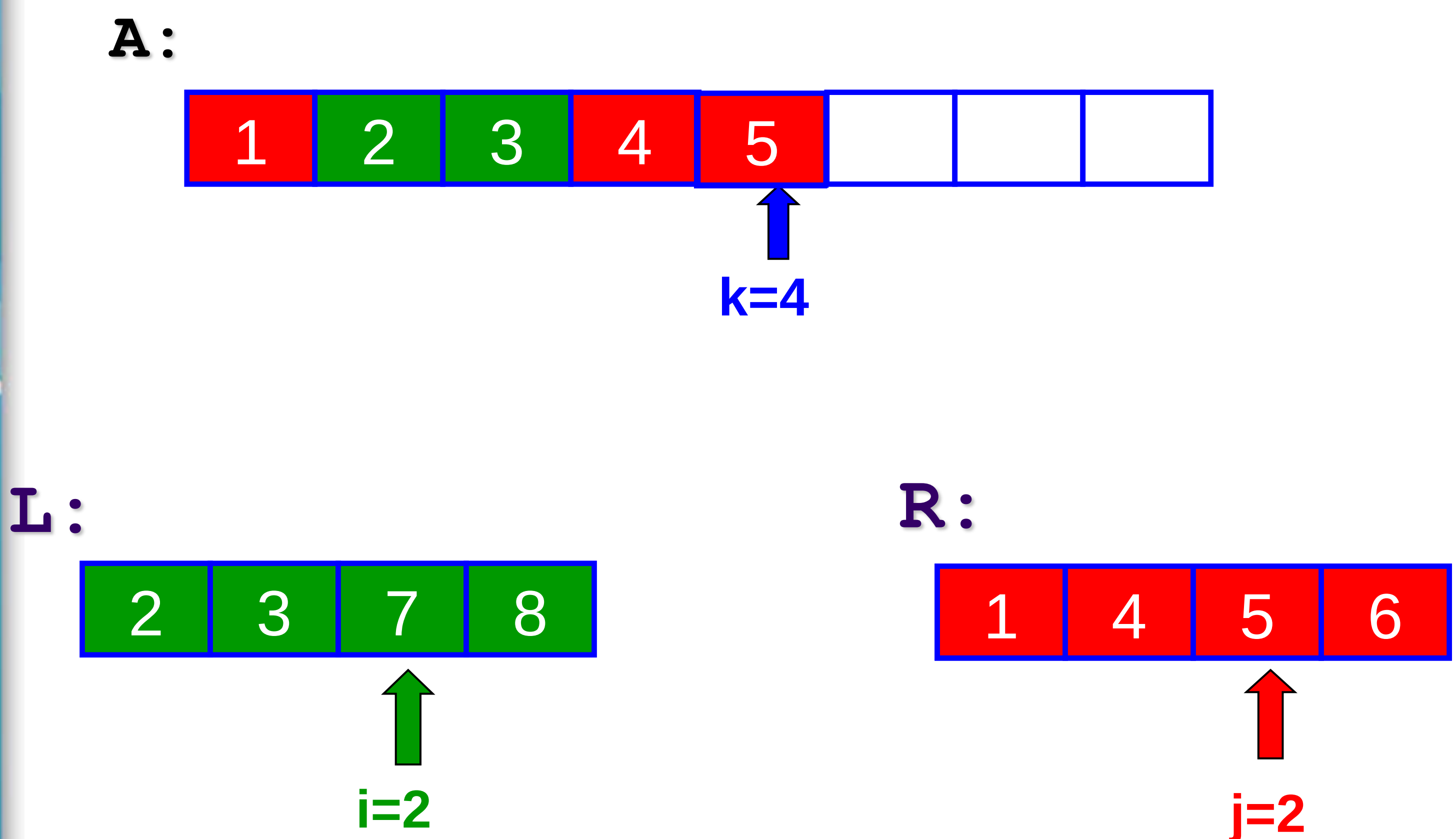
Merge-Sort: Merge Example



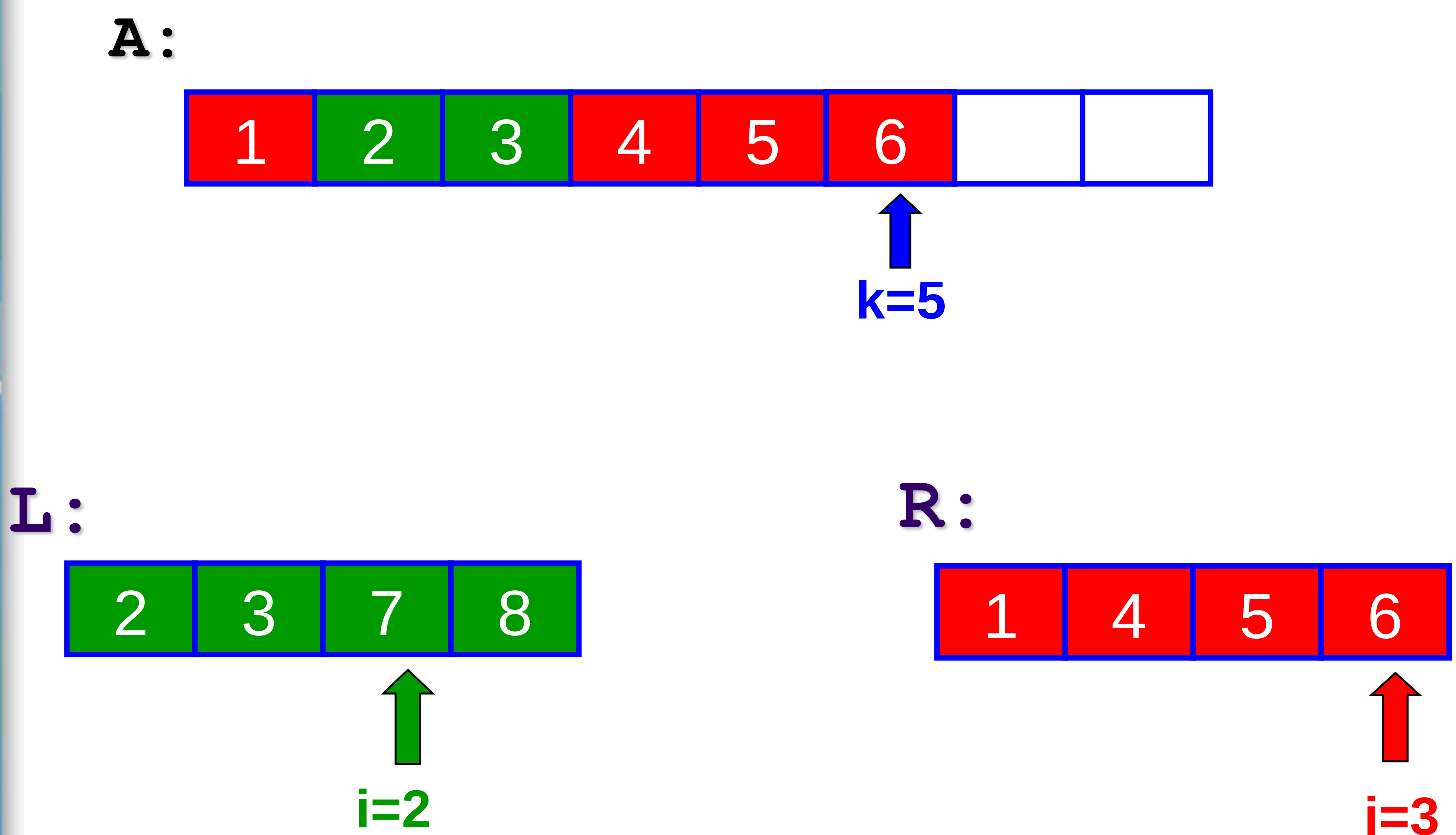
Merge-Sort: Merge Example



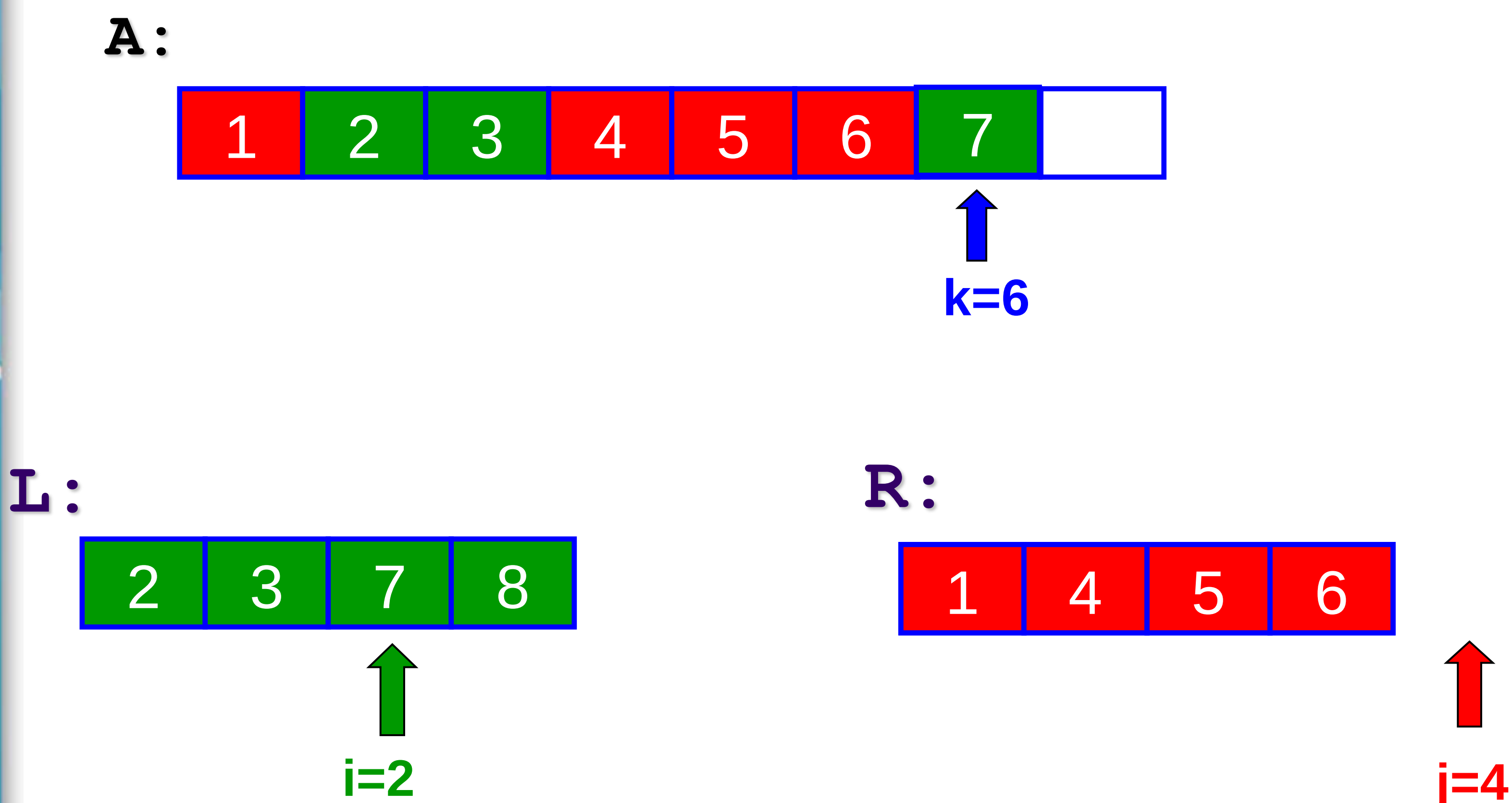
Merge-Sort: Merge Example



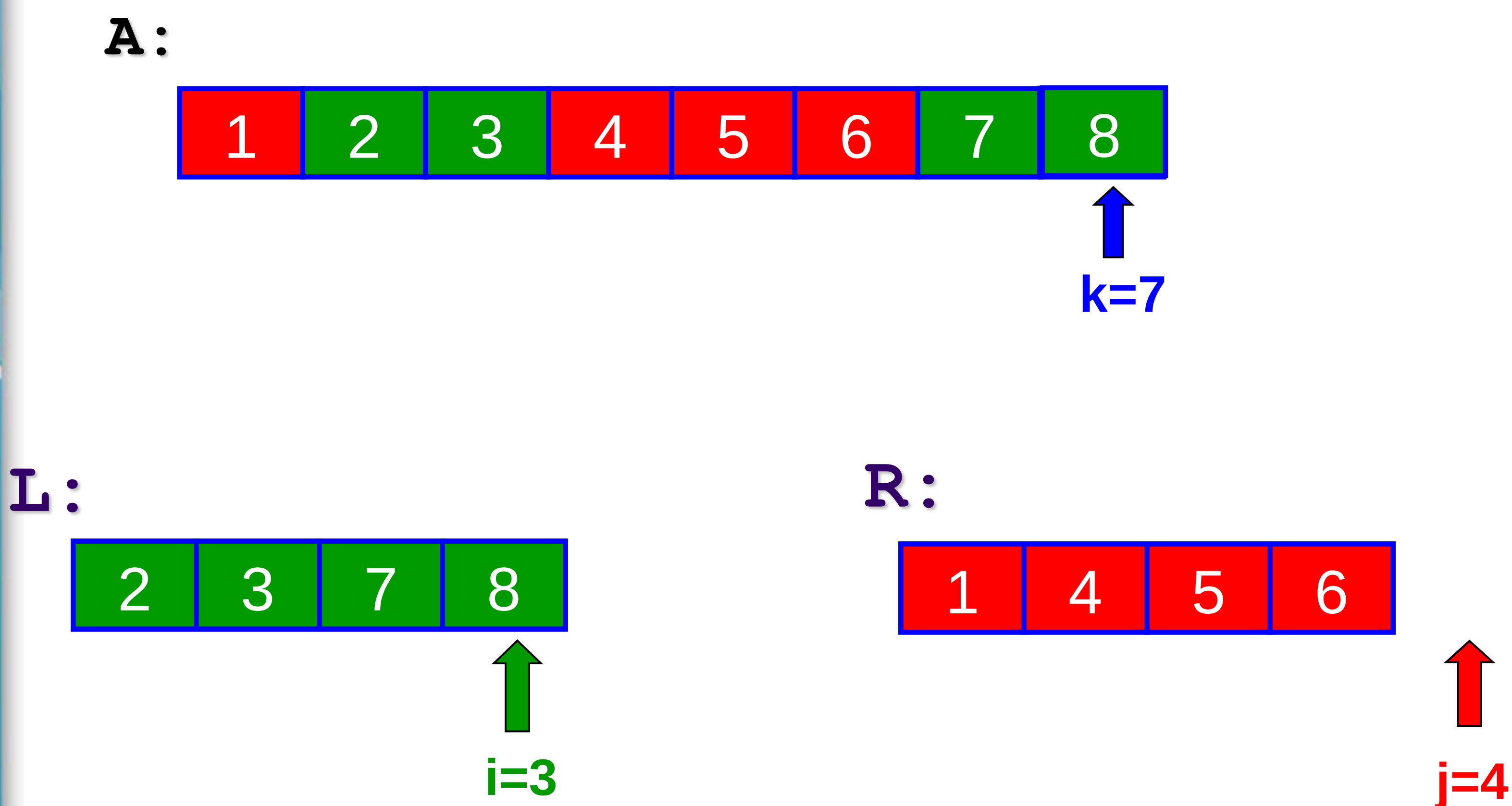
Merge-Sort: Merge Example



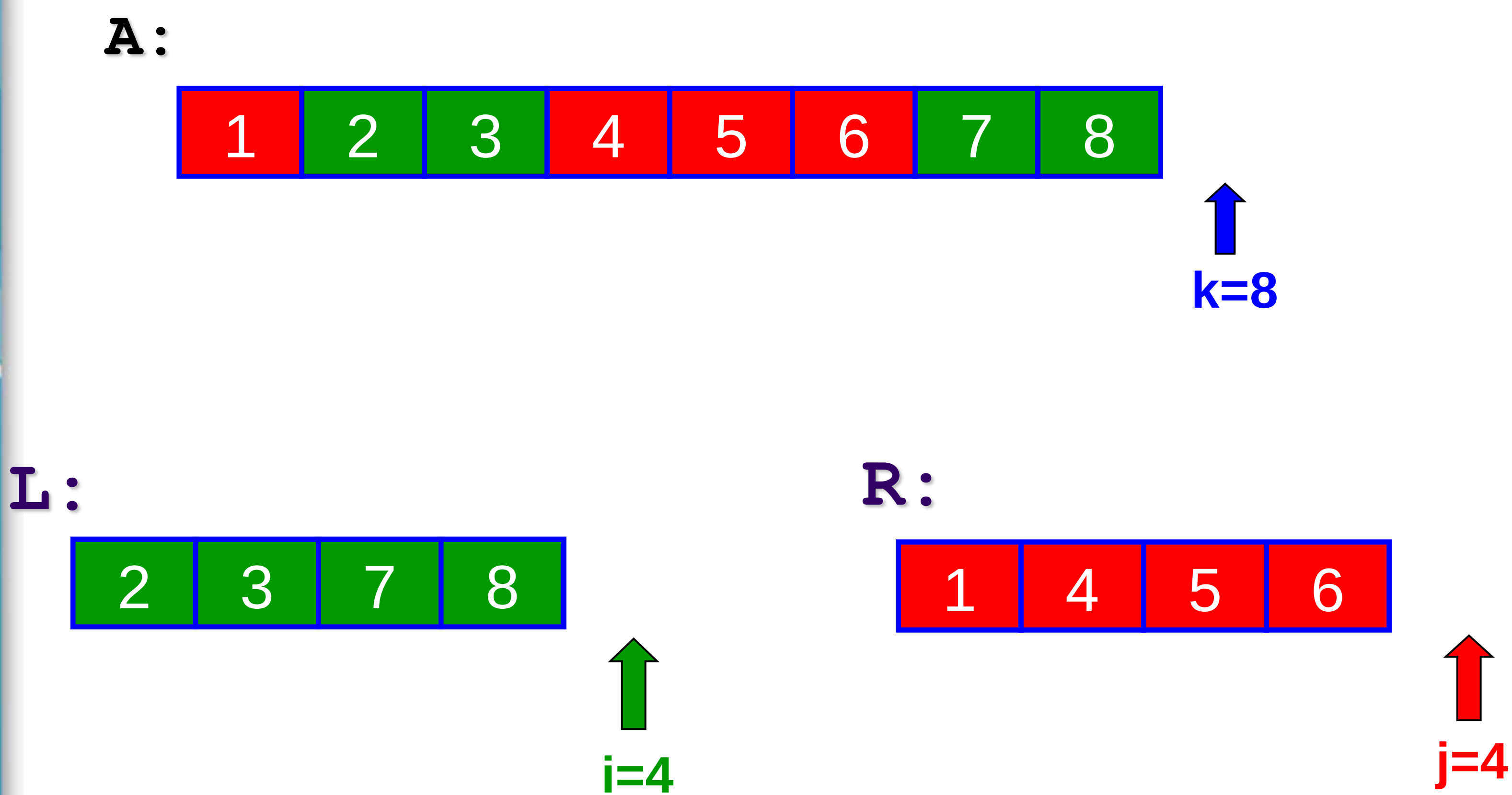
Merge-Sort: Merge Example



Merge-Sort: Merge Example



Merge-Sort: Merge Example



MERGE SORT ILLUSTRATION

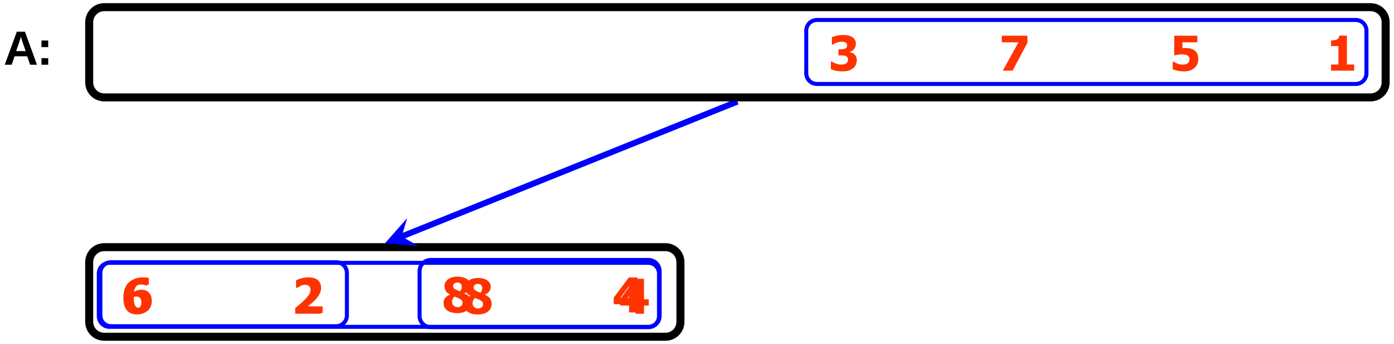
Merge-Sort(A, 0, 7)

Divide

A: 6 2 8 4 3 3 7 7 5 5 1 1

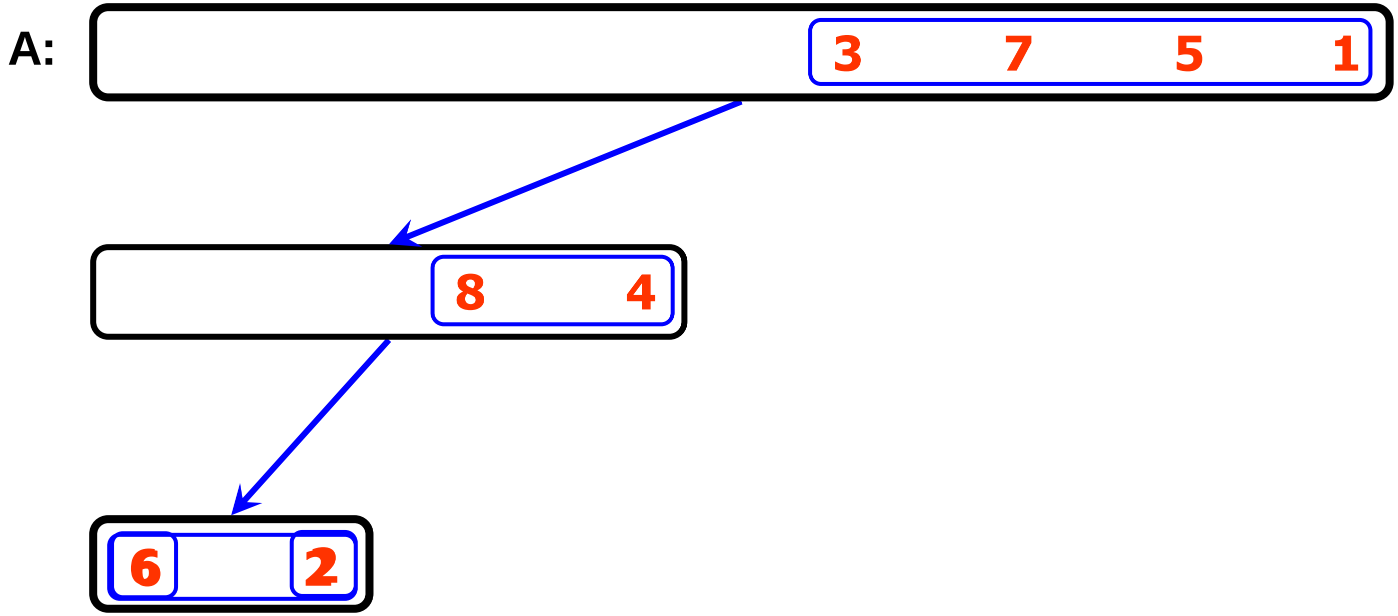
Merge-Sort(A, 0, 7)

Merge-Sort(A, 0, 3) , divide



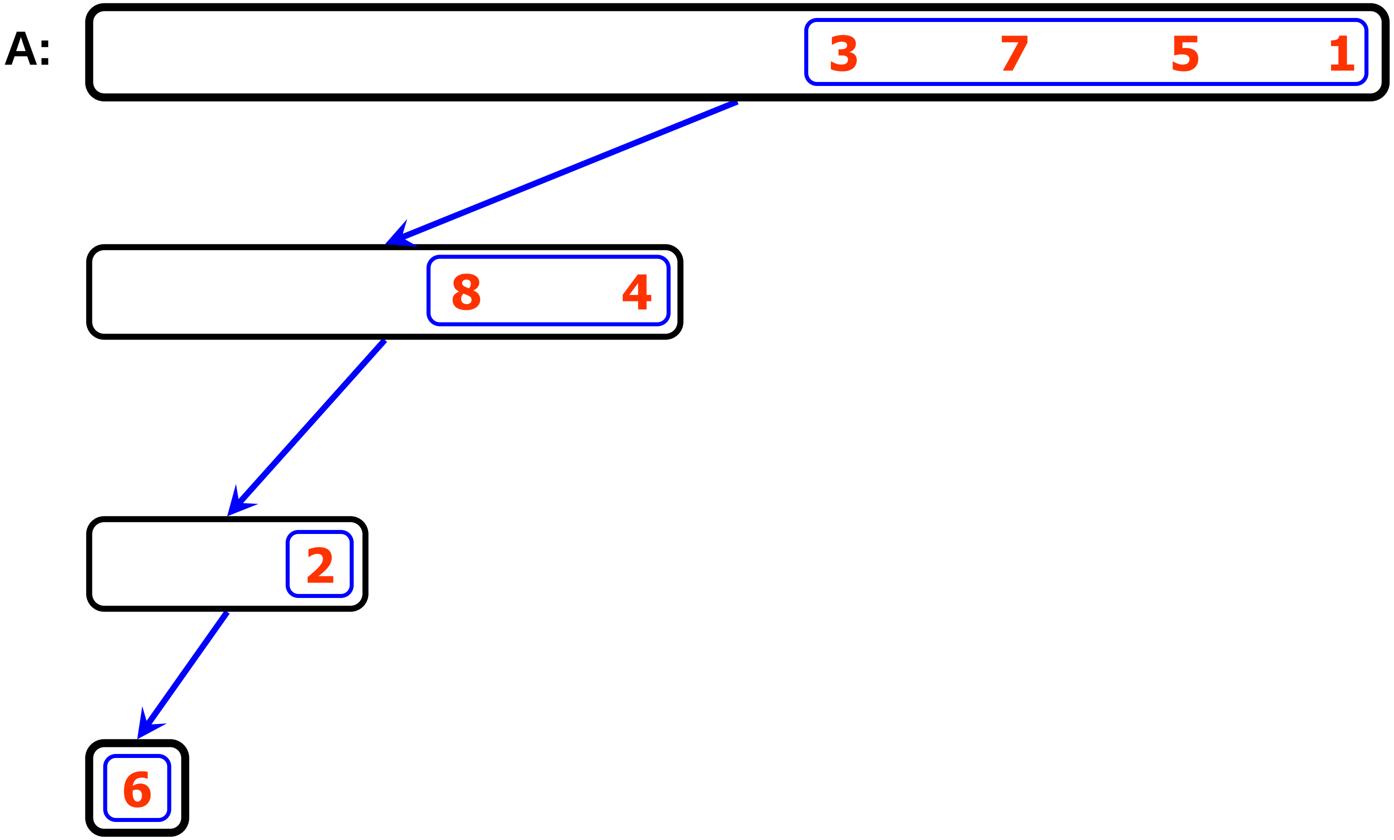
Merge-Sort(A, 0, 7)

Merge-Sort(A, 0, 1), divide



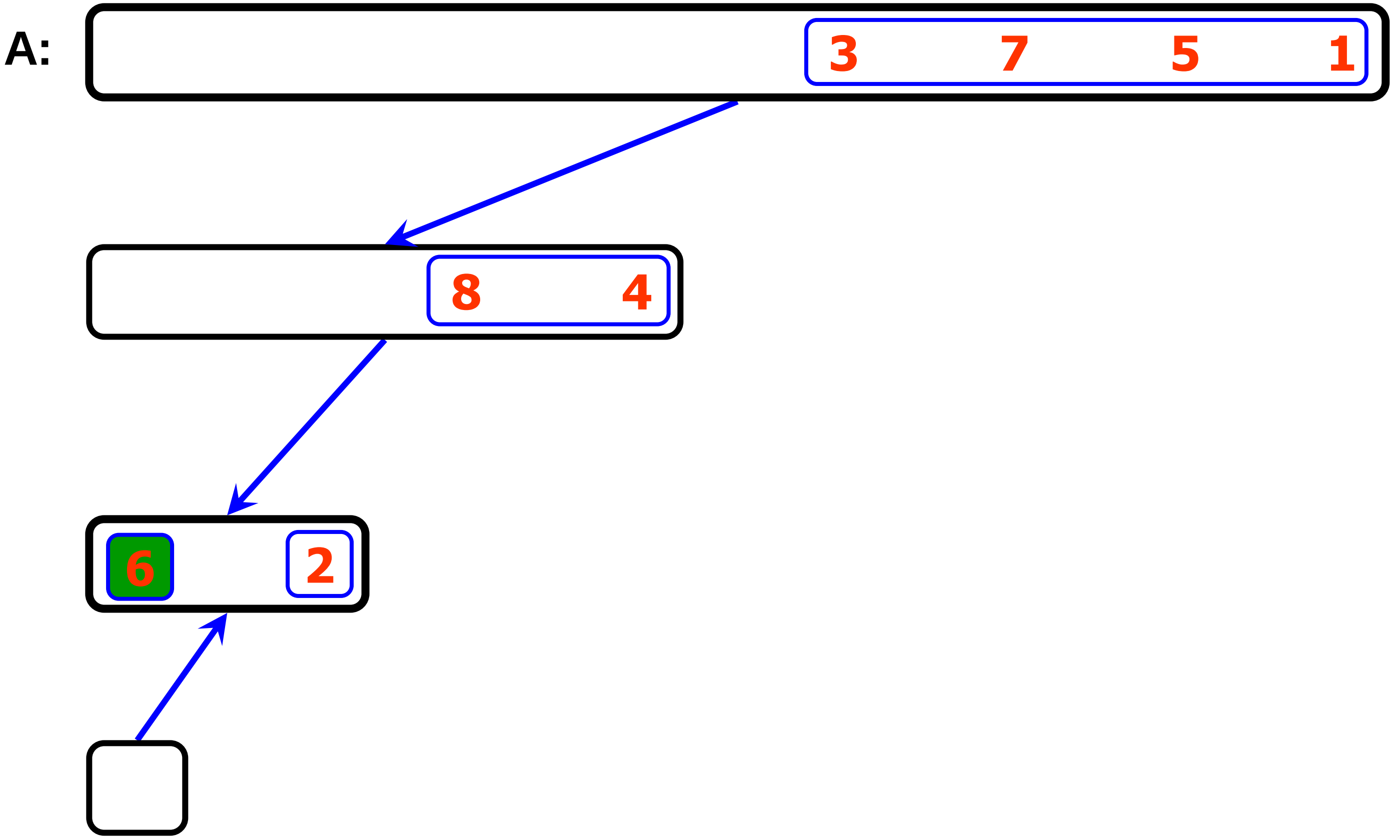
Merge-Sort(A, 0, 7)

Merge-Sort(A, 0, 0) , base case



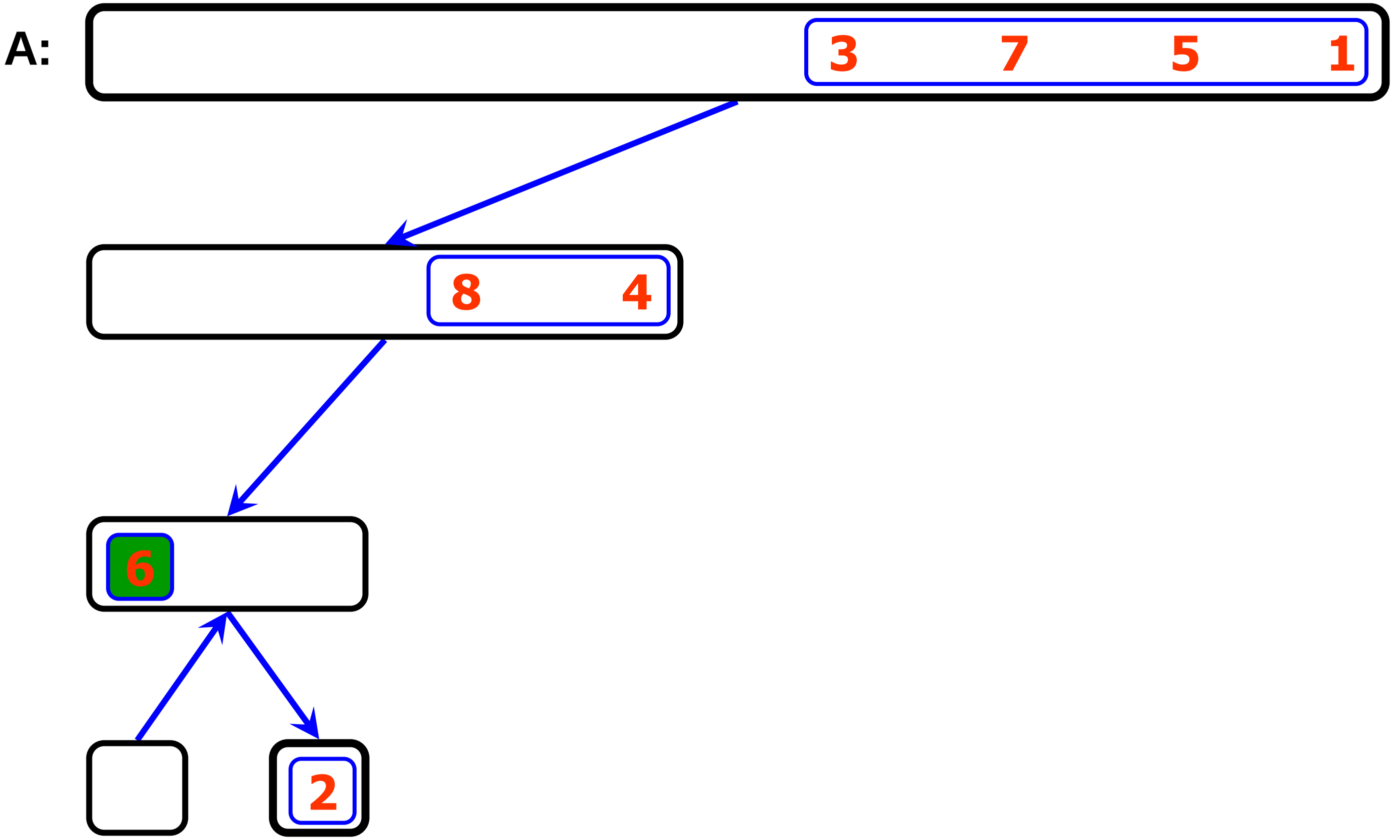
Merge-Sort(A, 0, 7)

Merge-Sort(A, 0, 0), return



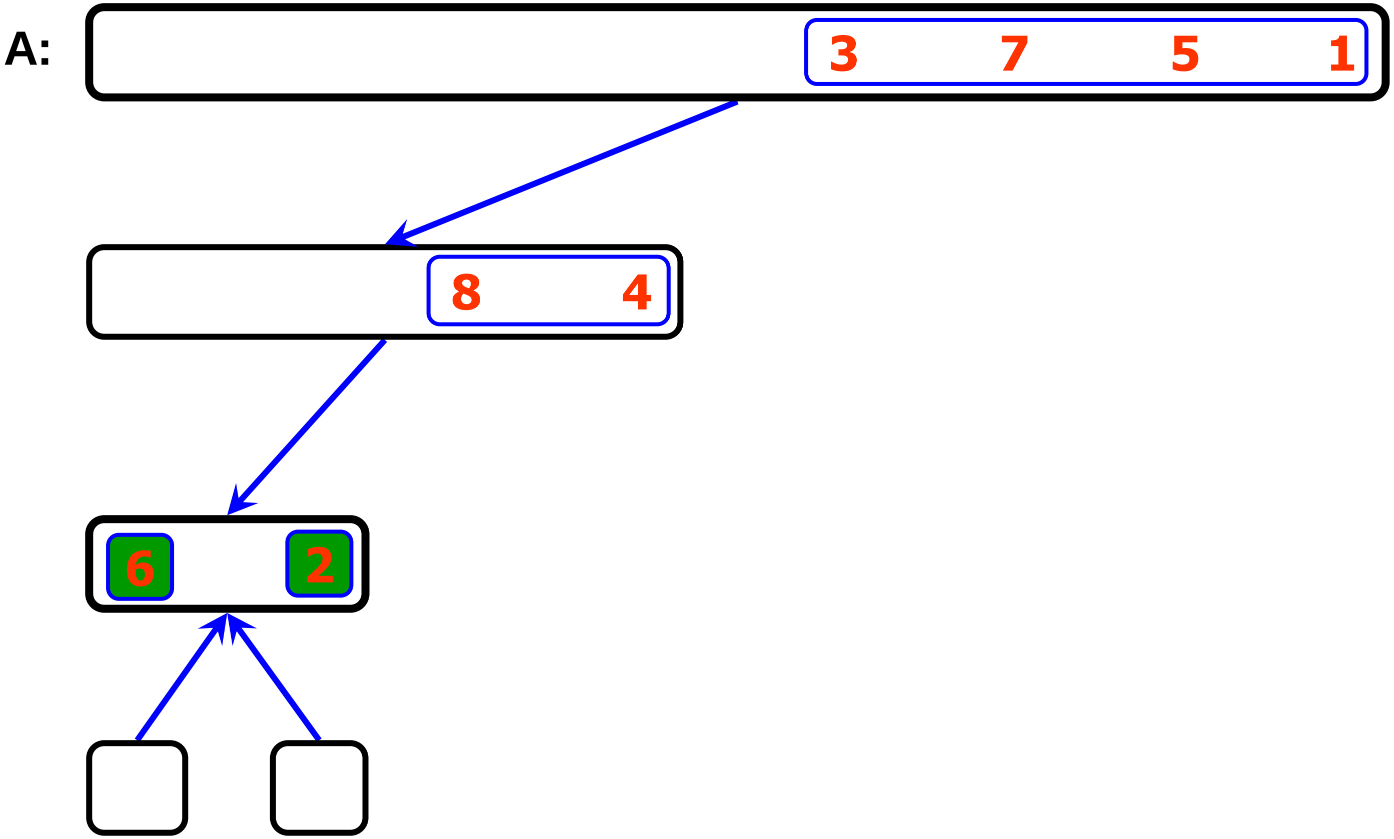
Merge-Sort(A, 0, 7)

Merge-Sort(A, 1, 1), base case



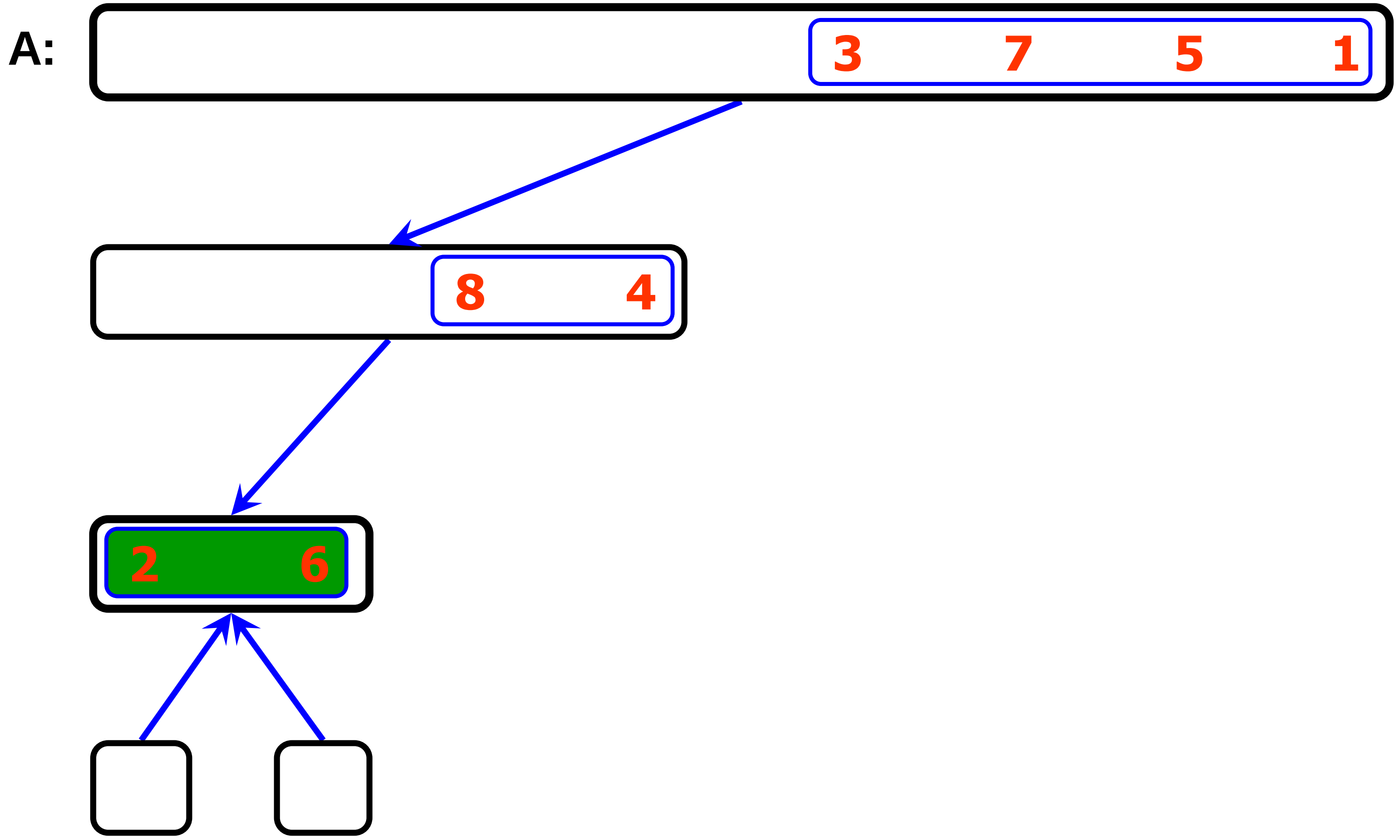
Merge-Sort(A, 0, 7)

Merge-Sort(A, 1, 1), return



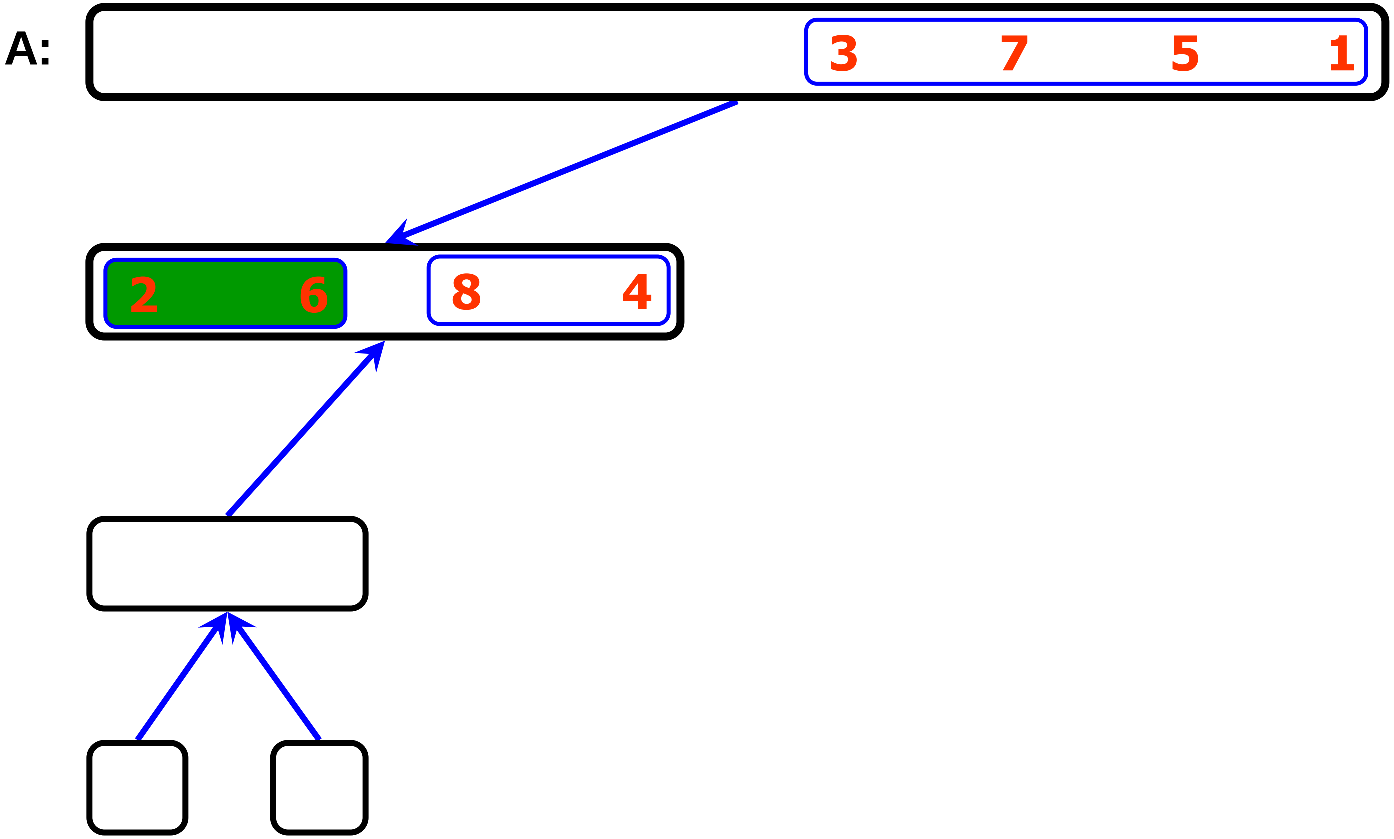
Merge-Sort(A, 0, 7)

Merge(A, 0, 0, 1)



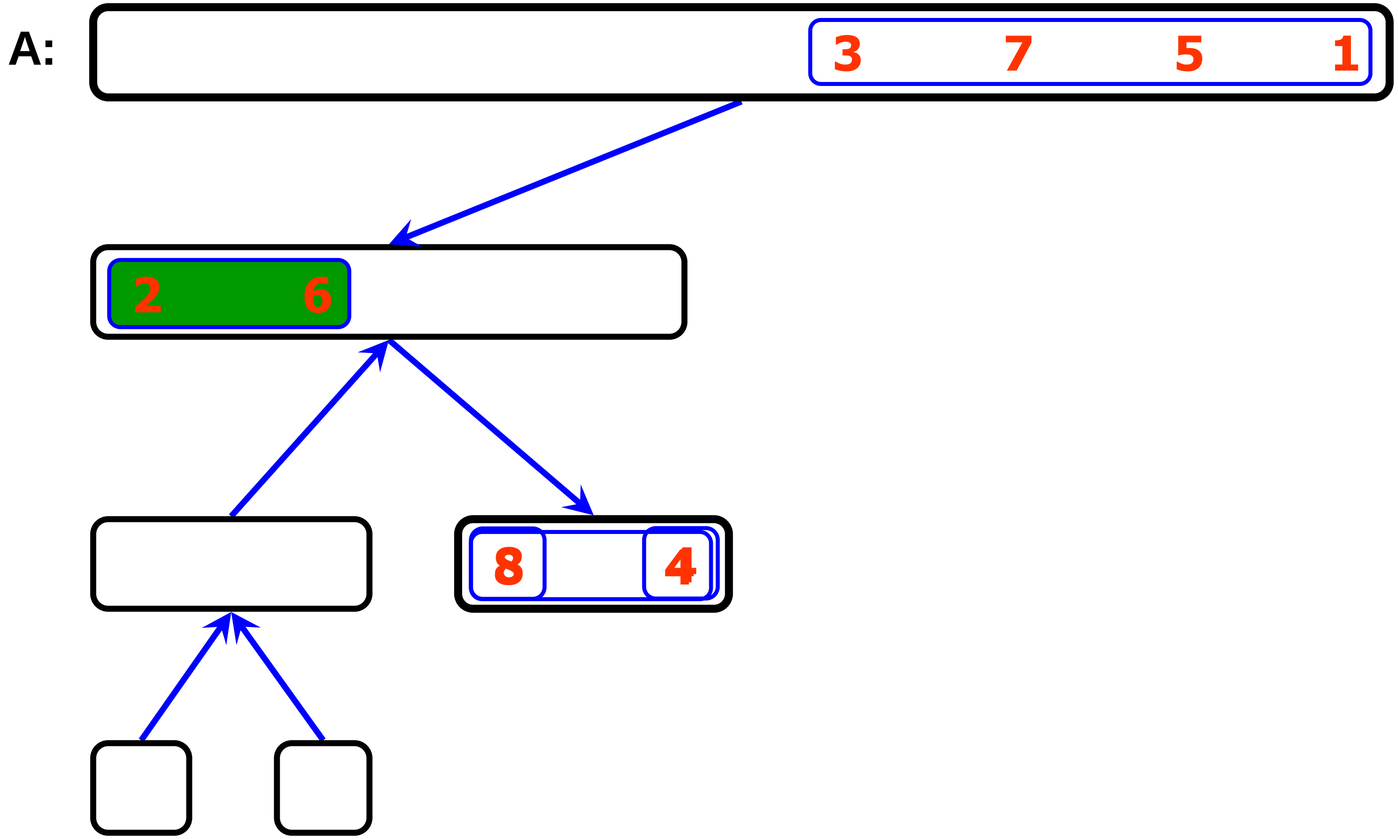
Merge-Sort(A, 0, 7)

Merge-Sort(A, 0, 1), return



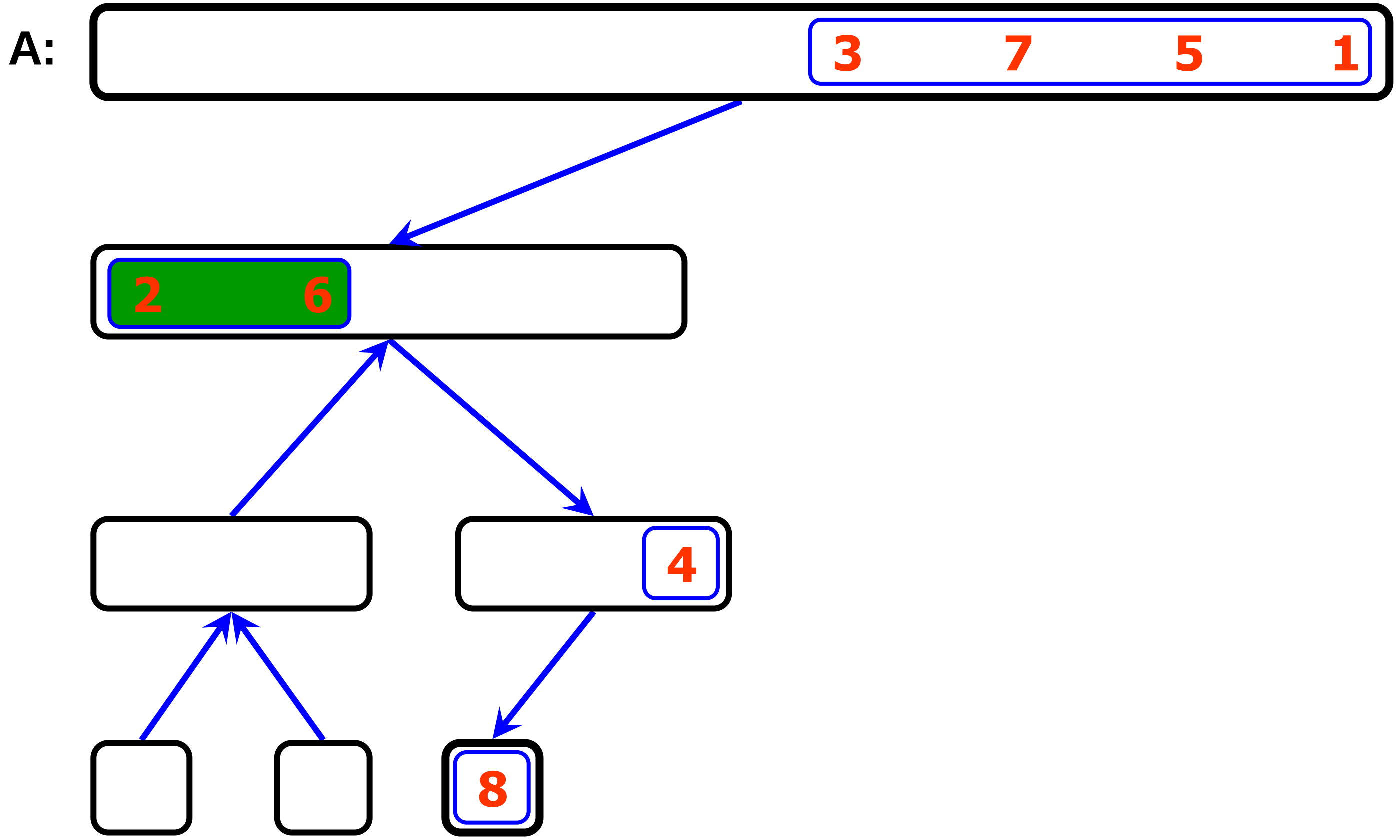
Merge-Sort(A, 0, 7)

Merge-Sort(A, 2, 3) , divide



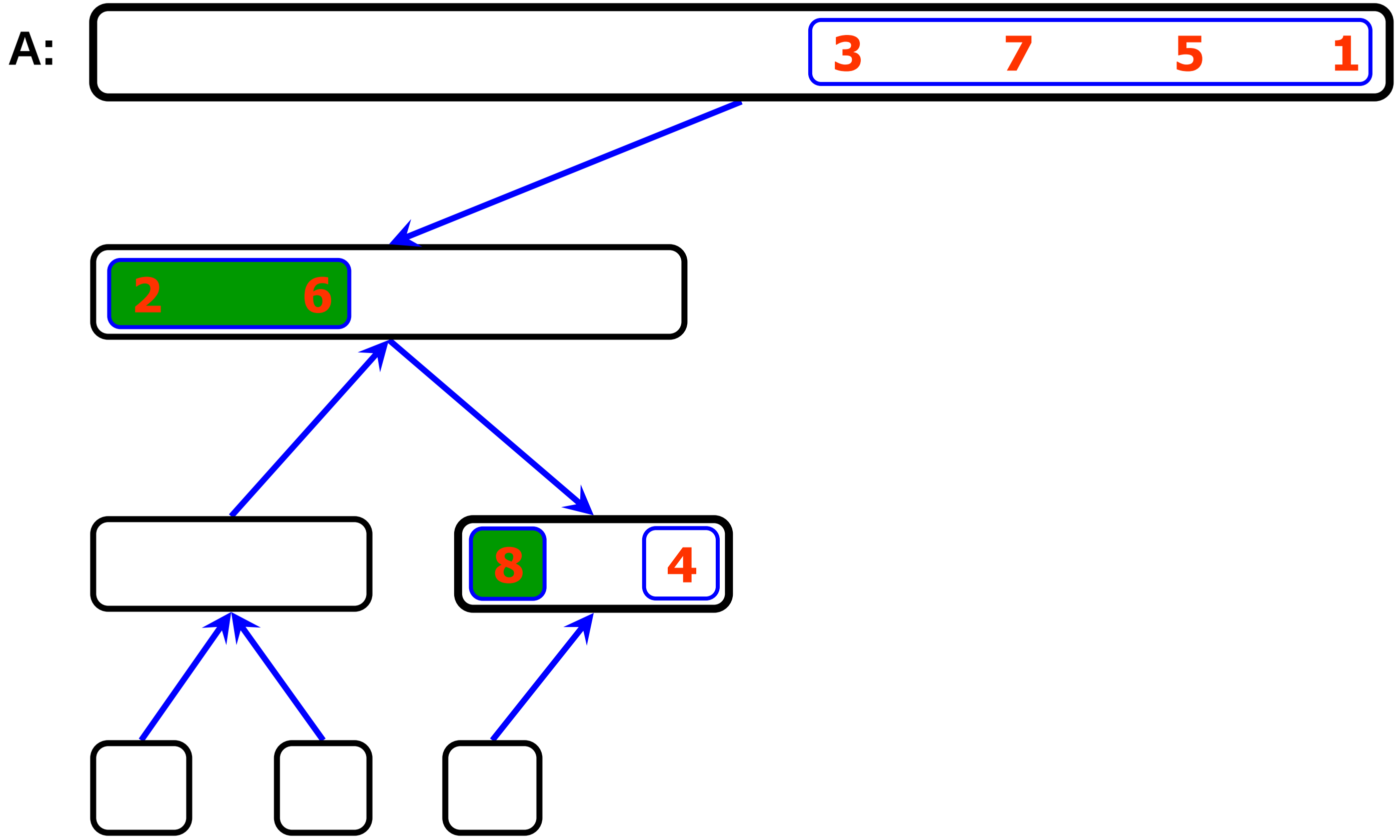
Merge-Sort(A, 0, 7)

Merge-Sort(A, 2, 2), base case



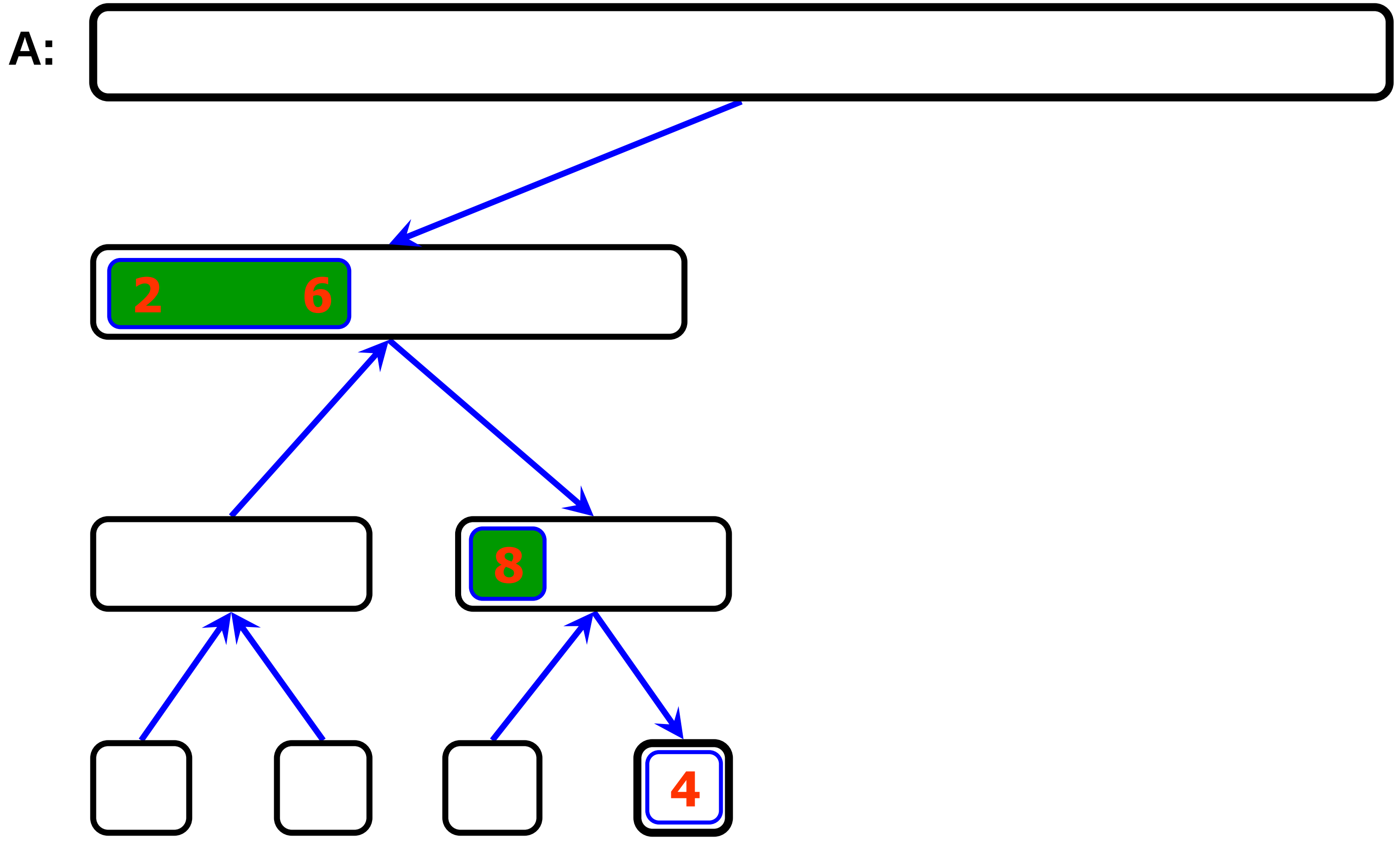
Merge-Sort(A, 0, 7)

Merge-Sort(A, 2, 2), return



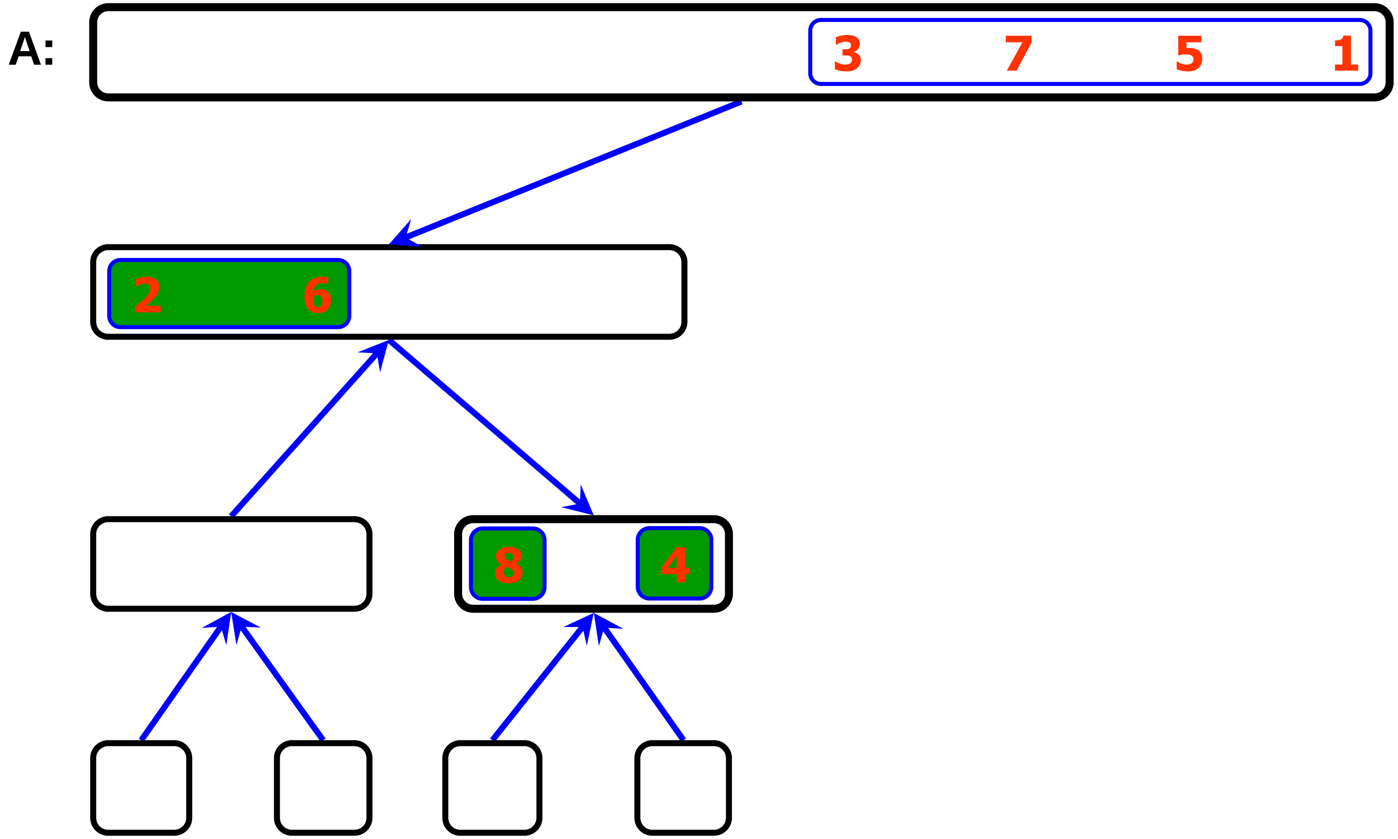
Merge-Sort(A, 0, 7)

Merge-Sort(A, 3, 3), base case



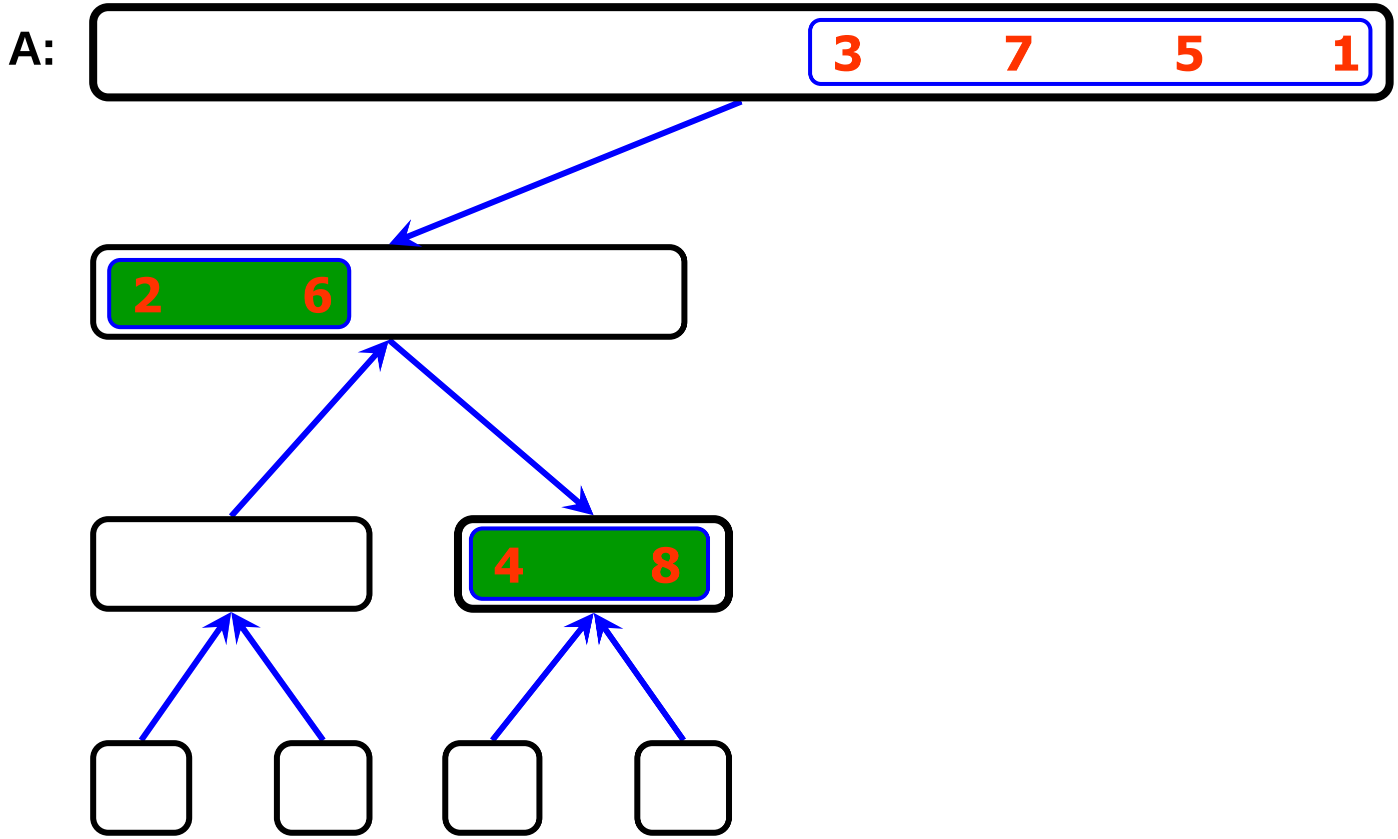
Merge-Sort(A, 0, 7)

Merge-Sort(A, 3, 3), return



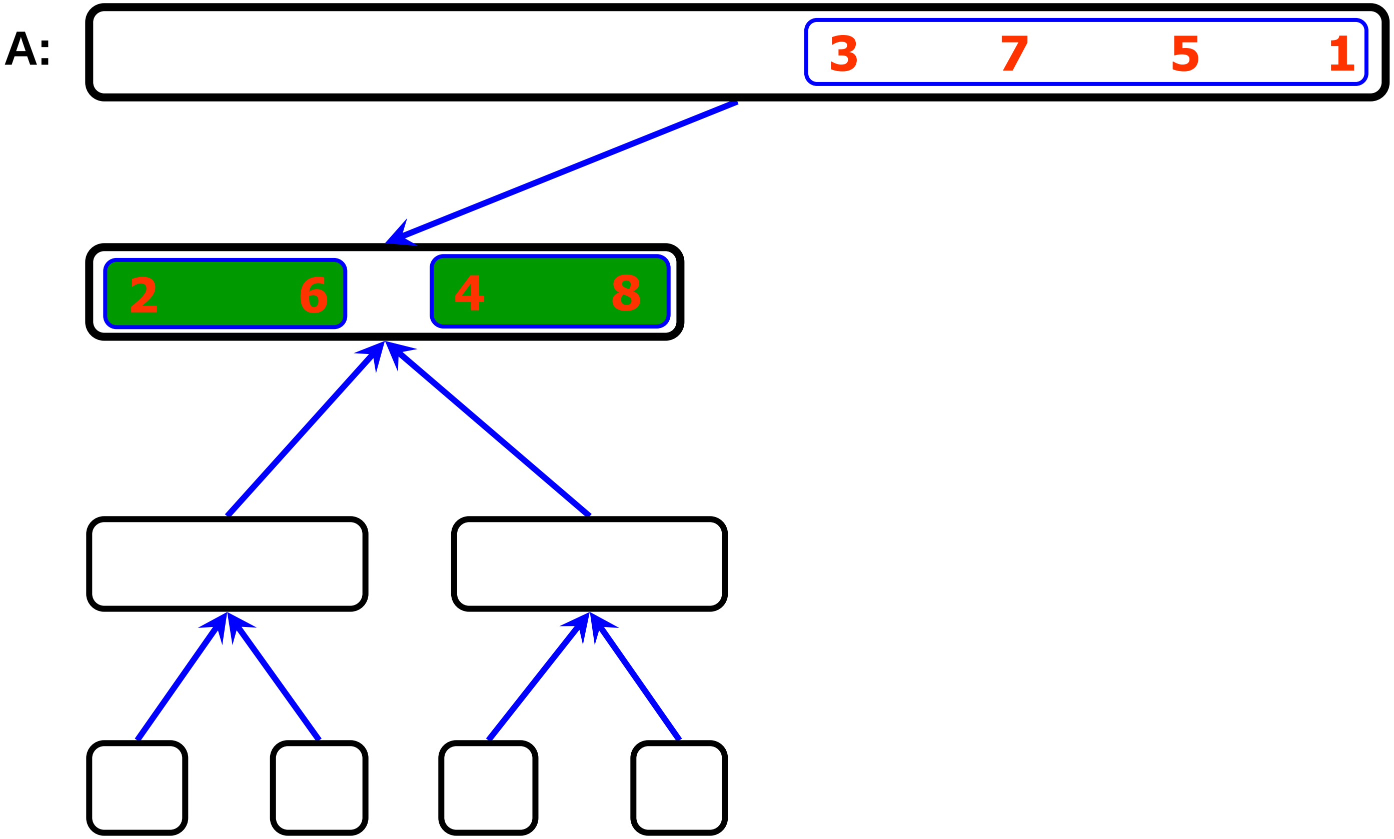
Merge-Sort(A, 0, 7)

Merge(A, 2, 2, 3)



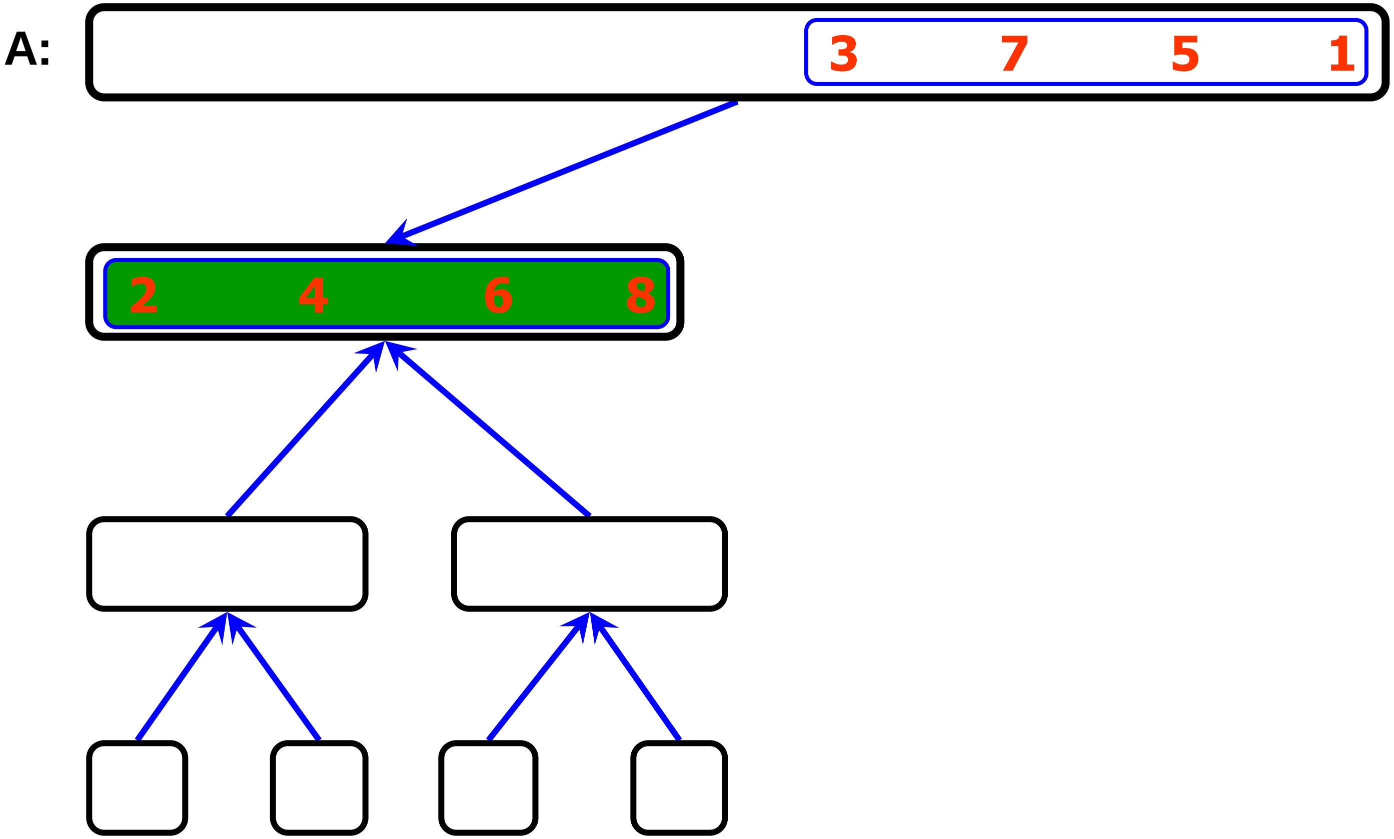
Merge-Sort(A, 0, 7)

Merge-Sort(A, 2, 3), return



Merge-Sort(A, 0, 7)

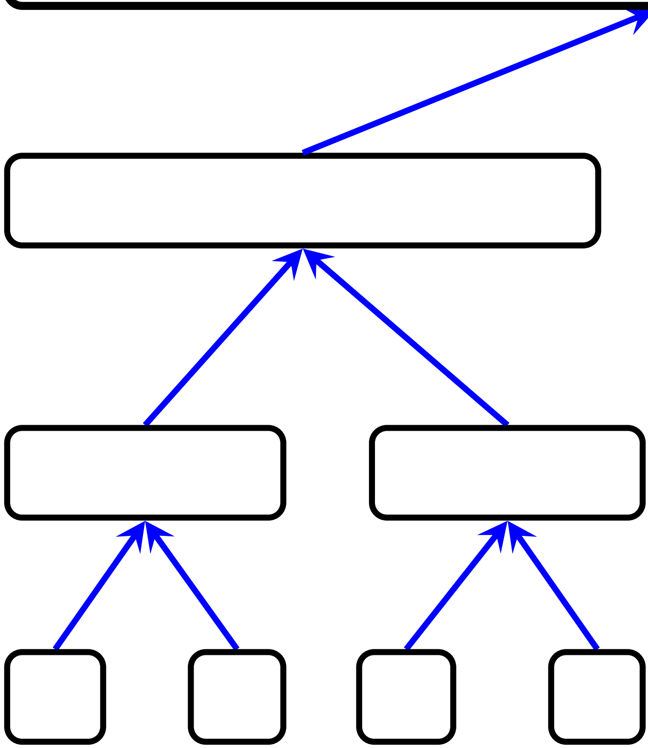
Merge(A, 0, 1, 3)



Merge-Sort(A, 0, 7)

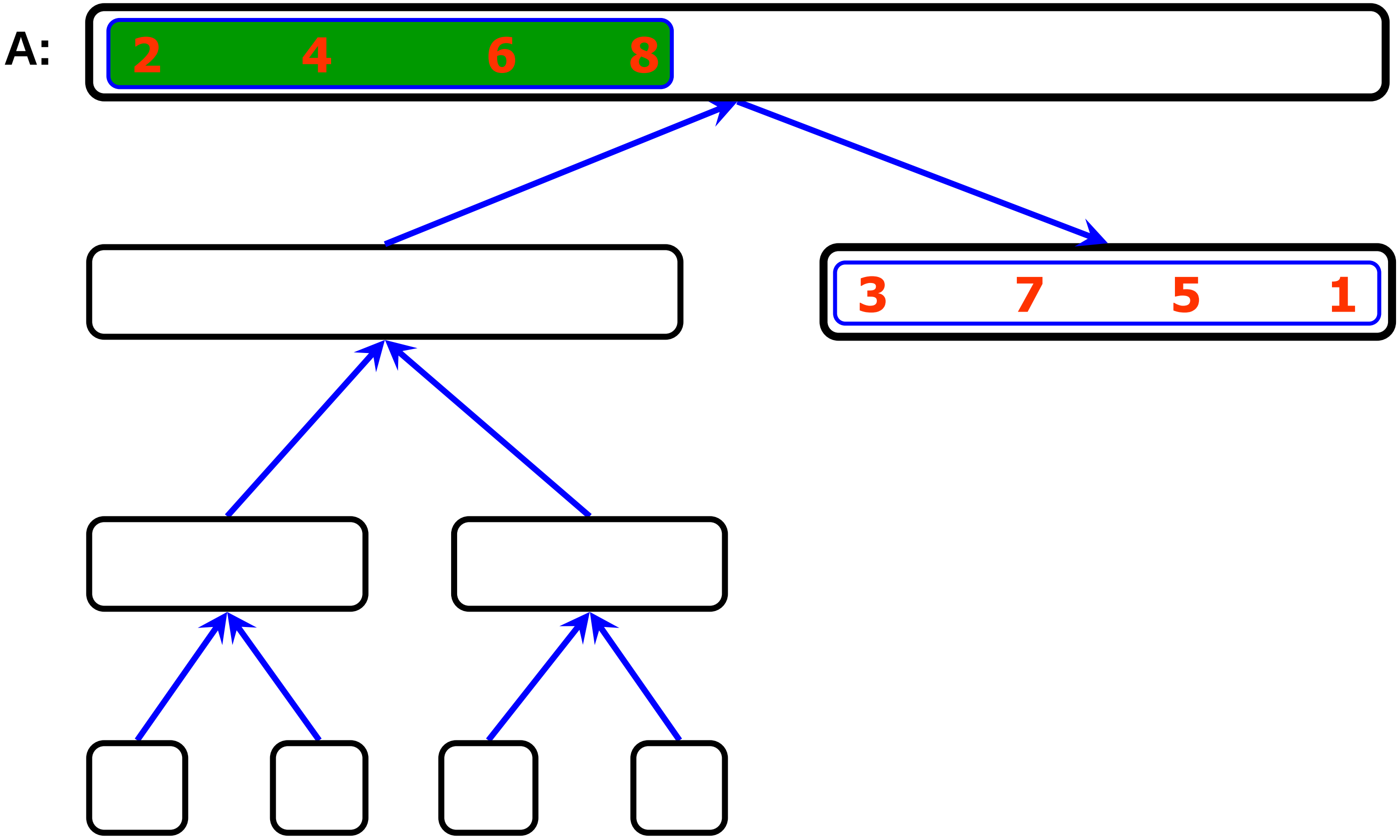
Merge-Sort(A, 0, 3), return

A: 2 4 6 8 3 7 5 1



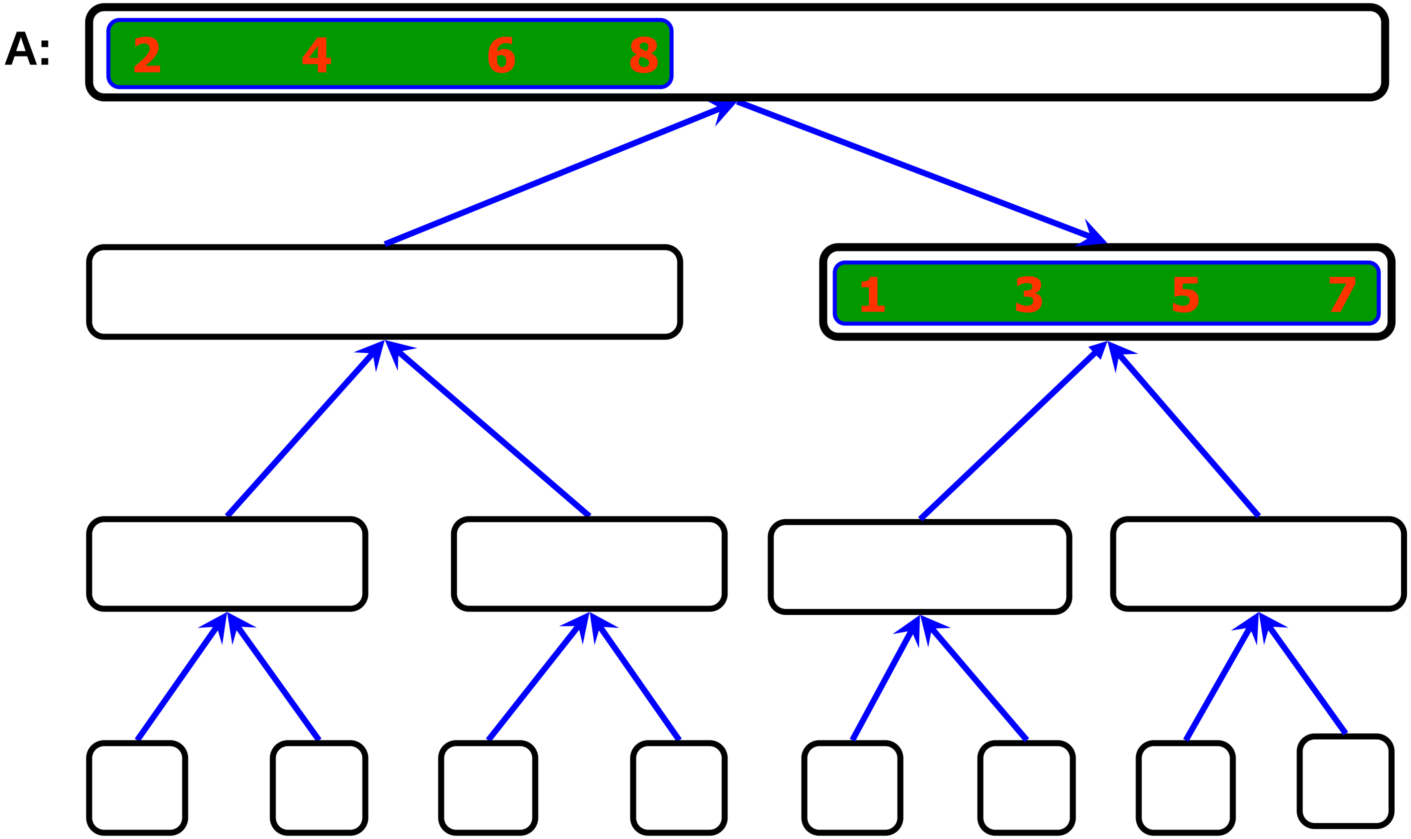
Merge-Sort(A, 0, 7)

Merge-Sort(A, 4, 7)



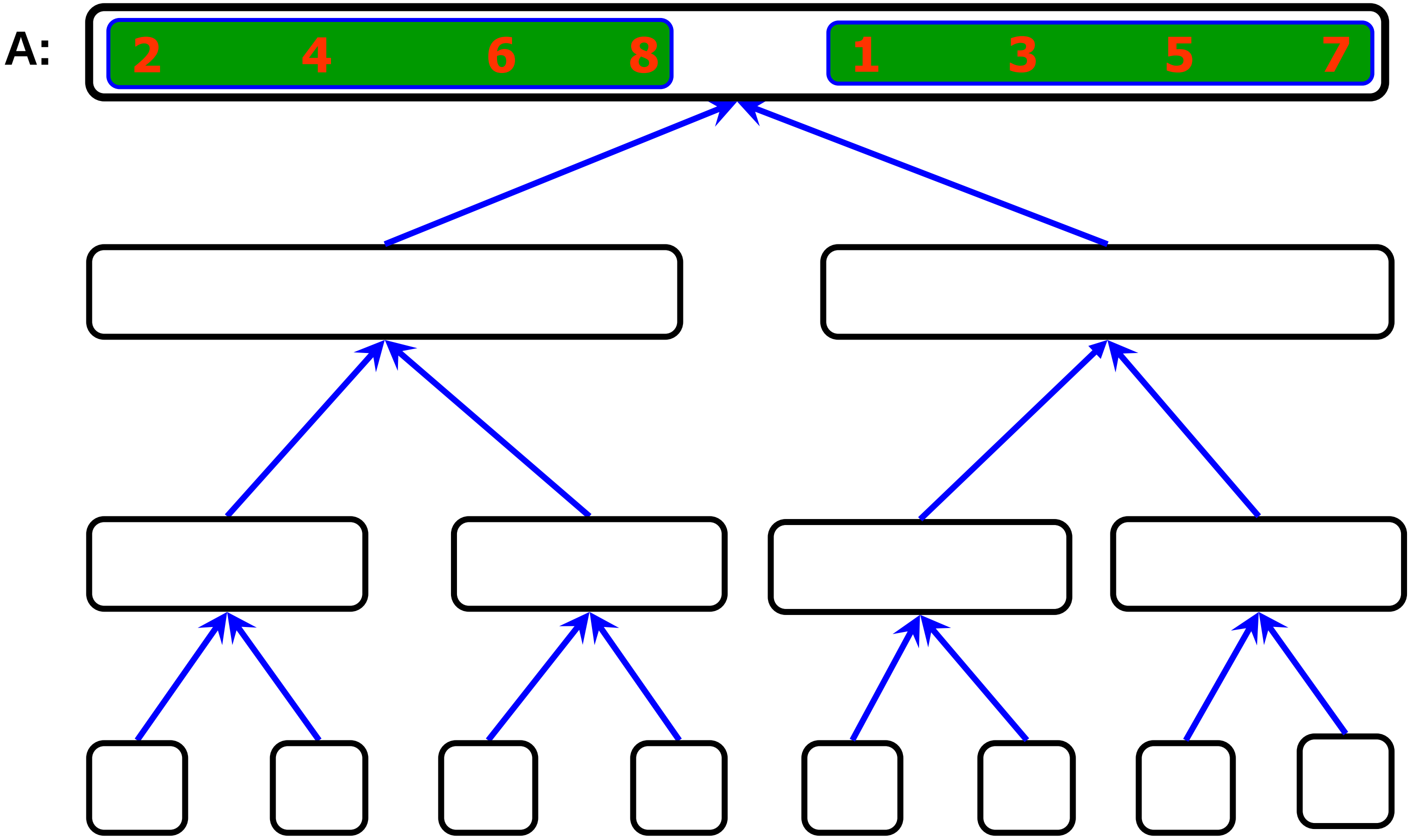
Merge-Sort(A, 0, 7)

Merge (A, 4, 5, 7)



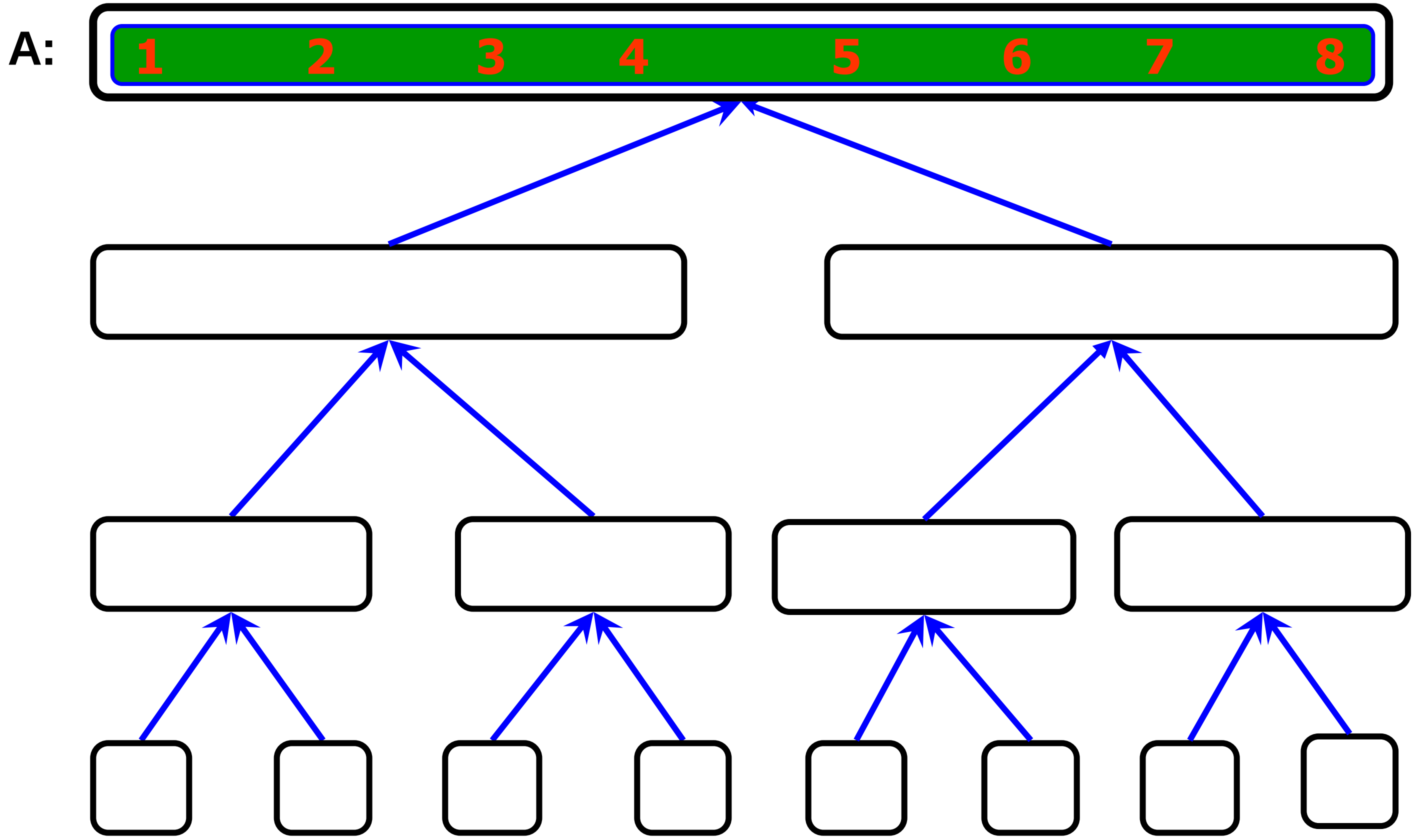
Merge-Sort(A, 0, 7)

Merge-Sort(A, 4, 7), return



Merge-Sort(A, 0, 7)

Merge-Sort(A, 0, 7), done!



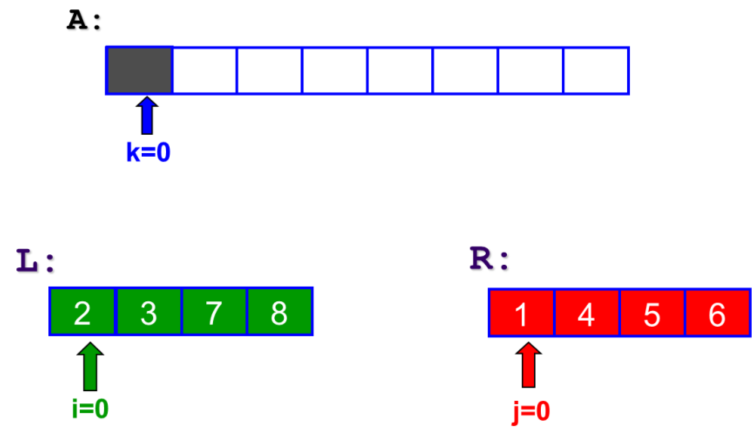
Merge Sort Algorithm and Coding

mergeSort

```
def mergeSort(arr):
        if len(arr) > 1:
                # Finding the mid of the array
                mid = len(arr)//2
                # Dividing the array elements
                L = arr[:mid]
                R = arr[mid:]
                # Sorting the first half
                mergeSort(L)
                # Sorting the second half
                mergeSort(R)
```

demo: MergeSort.py

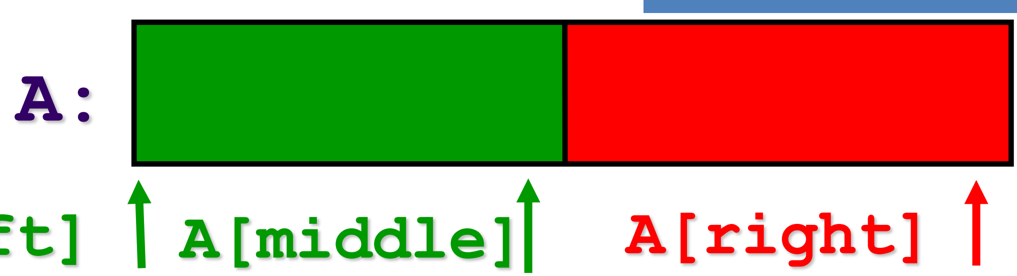
```
Merge(A, left, middle, right)
n1 ← middle - left + 1
n2 ← right - middle
create array L[n1], R[n2]
for i ← 0 to n1-1 do L[i] ← A[left +i]
for j ← 0 to n2-1 do R[j] ← A[middle+j+1]
k ← left
i ← j ← 0
while i < n1 &  j < n2
    if L[i] < R[j]
        A[k++] ← L[i++]
    else
        A[k++] ← R[j++]
while i < n1
    A[k++] ← L[i++]
while j < n2
    A[k++] ← R[j++]
```



$n = n_1+n_2$

Space: n

Time : cn for some constant c



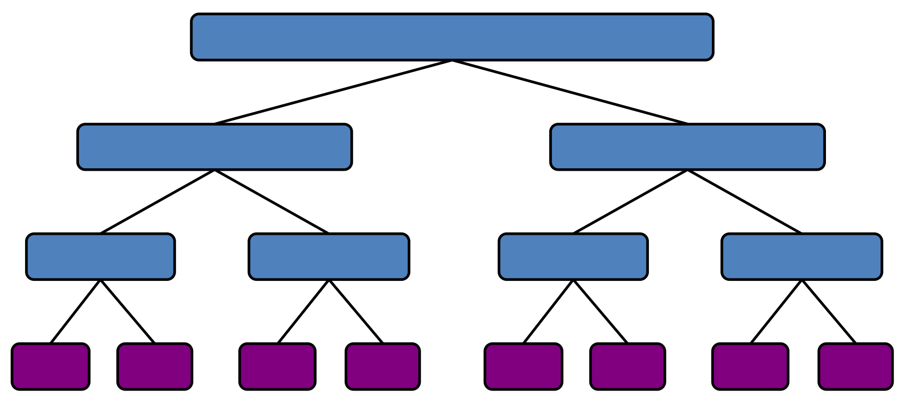
merge

```
 # Copy data to temp arrays L[] and R[]
while i < len(L) and j < len(R):
         if L[i] < R[j]:
                  arr[k] = L[i]
                  i += 1
         else:
                  arr[k] = R[j]
                  j += 1
         k += 1

# Checking if any element was left
while i < len(L):
         arr[k] = L[i]
         i += 1
         k += 1

while j < len(R):
         arr[k] = R[j]
         j += 1
         k += 1
```

Analysis of Merge-Sort

- The height h of the merge-sort tree is $O(\log n)$
  - at each recursive call we divide in half the sequence,
- The overall amount or work done at the nodes of depth i is $O(n)$
  - we partition and merge 2^i sequences of size $n/2^i$
  - we make 2^{i+1} recursive calls
- Thus, the total running time of merge-sort is $O(n \log n)$

depth	#seqs	size
0	1	n
1	2	$n/2$
i	2^i	$n/2^i$
…	…	…



QUICK SORT

Quick sort

- Quick sort is a highly efficient sorting algorithm and is based on partitioning of array of data into smaller arrays.
- A large array is partitioned into two arrays one of which holds values smaller than the specified value, say **pivot.**
- Based on *pivot* the partition is made and another array holds values greater than the pivot value.

Quick Sort

- **Divide**:
  - Pick any element **p** as the **pivot**, e.g, the first element
  - ***Partition*** the remaining elements into

    FirstPart, which contains all elements < p

    SecondPart, which contains all elements ≥ p

- **Recursively sort** the FirstPart and SecondPart

- **Combine**: no work is necessary since sorting is done in place

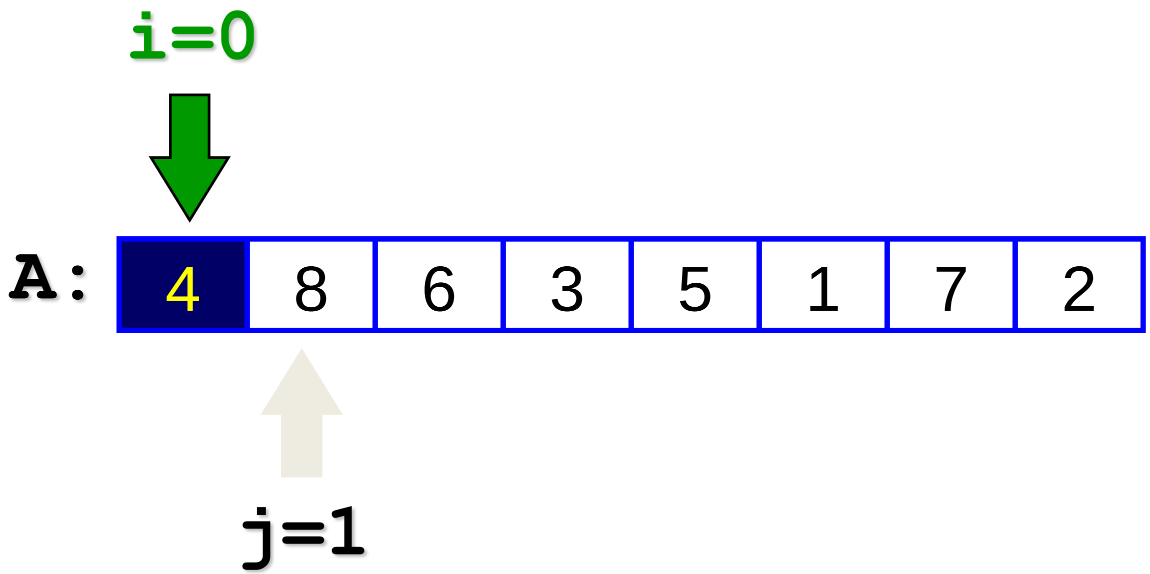
Partition in Array in Quick Sort

Partition Example

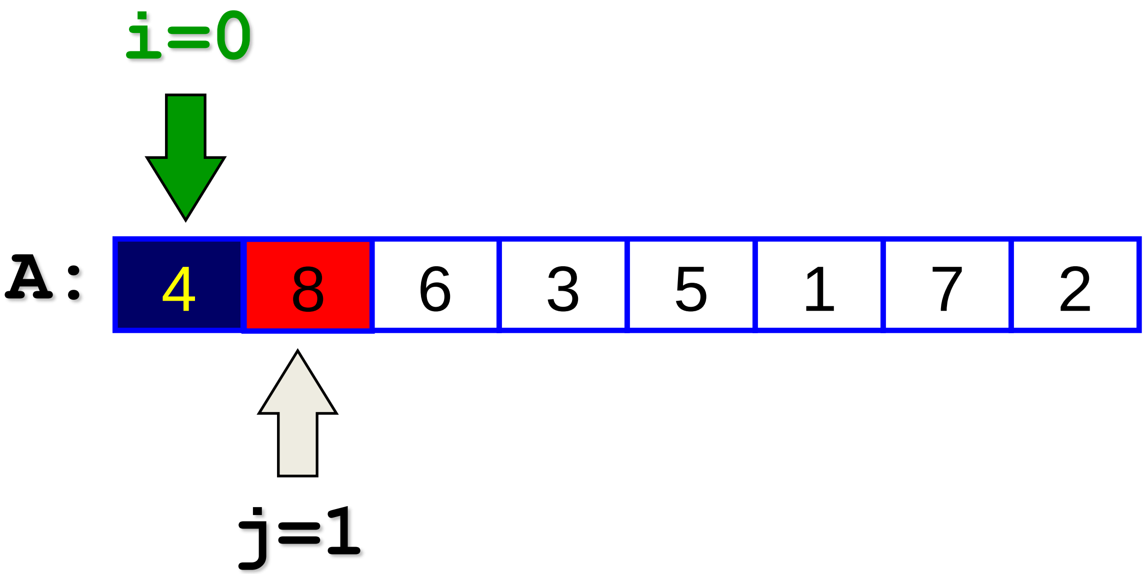
A:

4	8	6	3	5	1	7	2

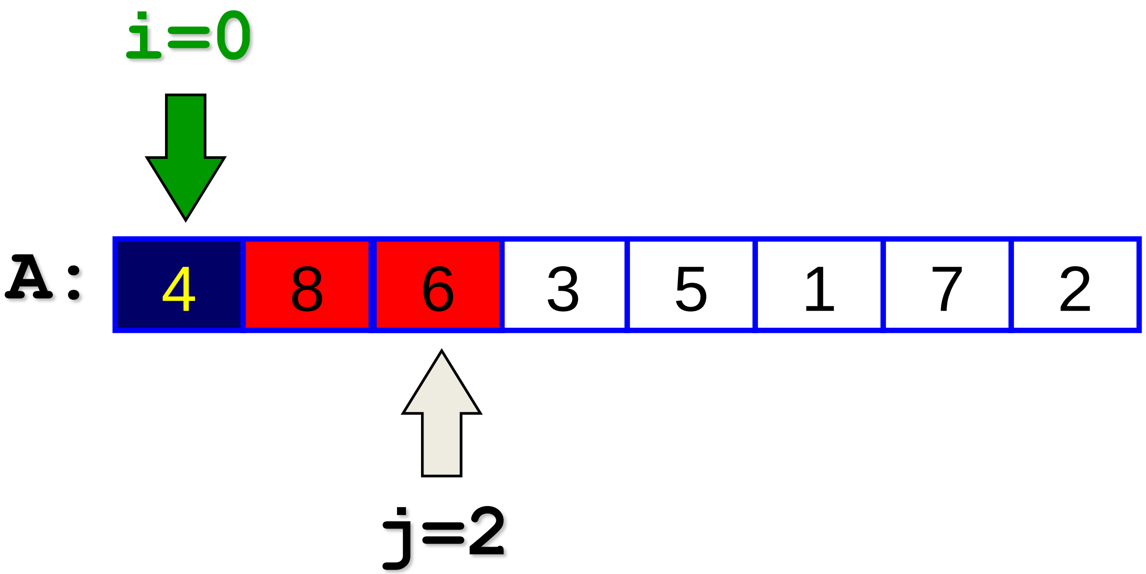
Partition Example



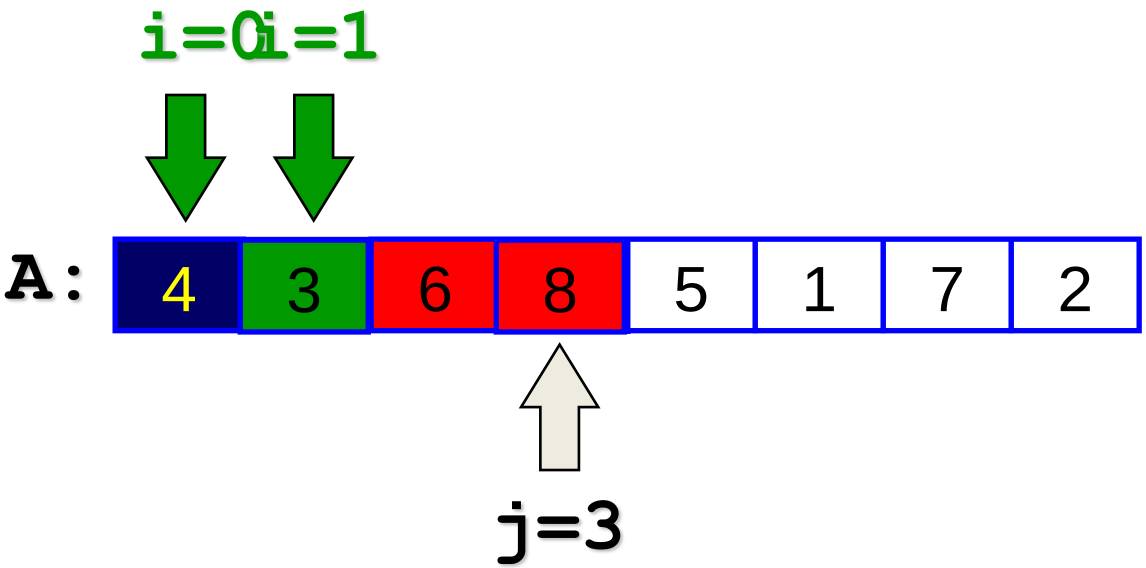
Partition Example



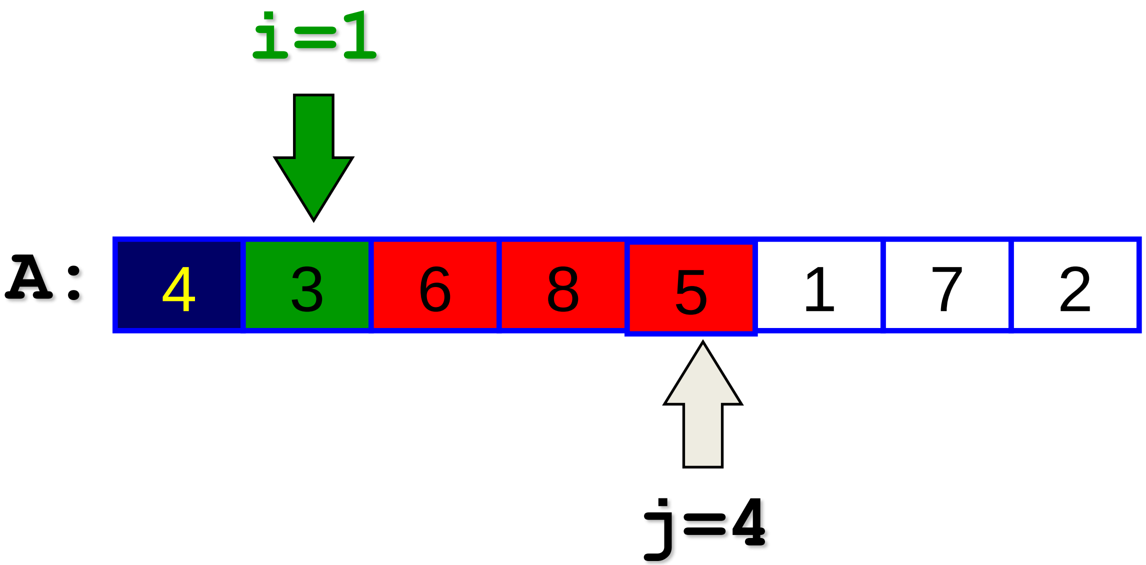
Partition Example



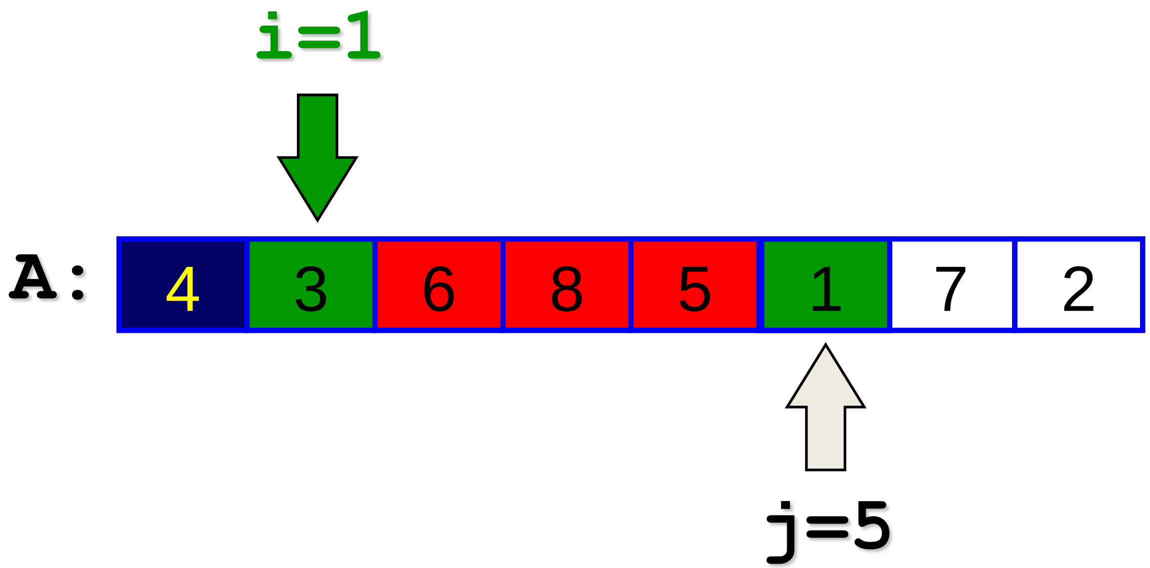
Partition Example



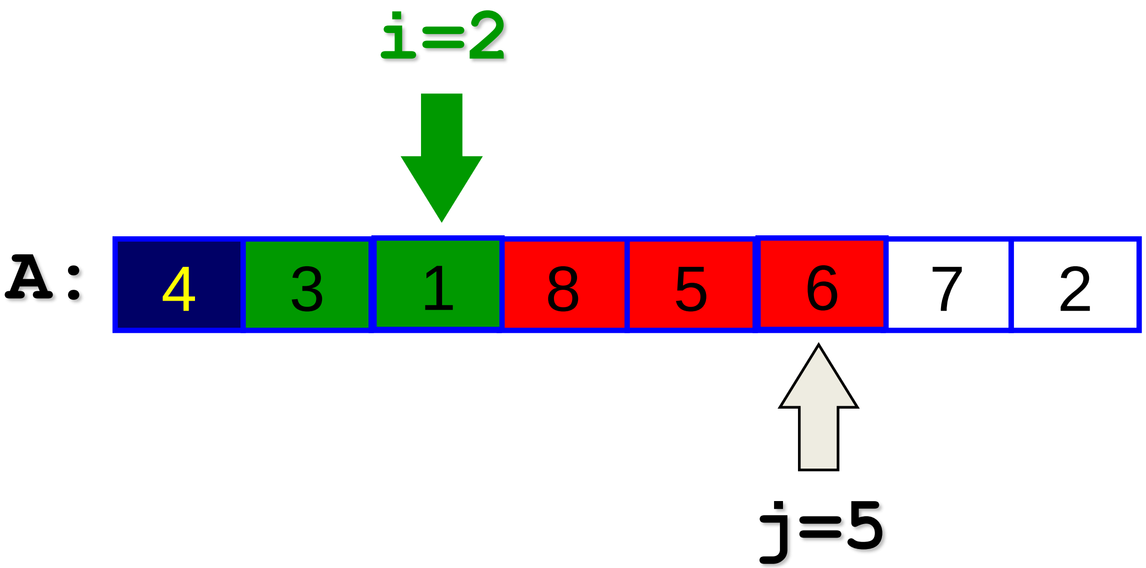
Partition Example



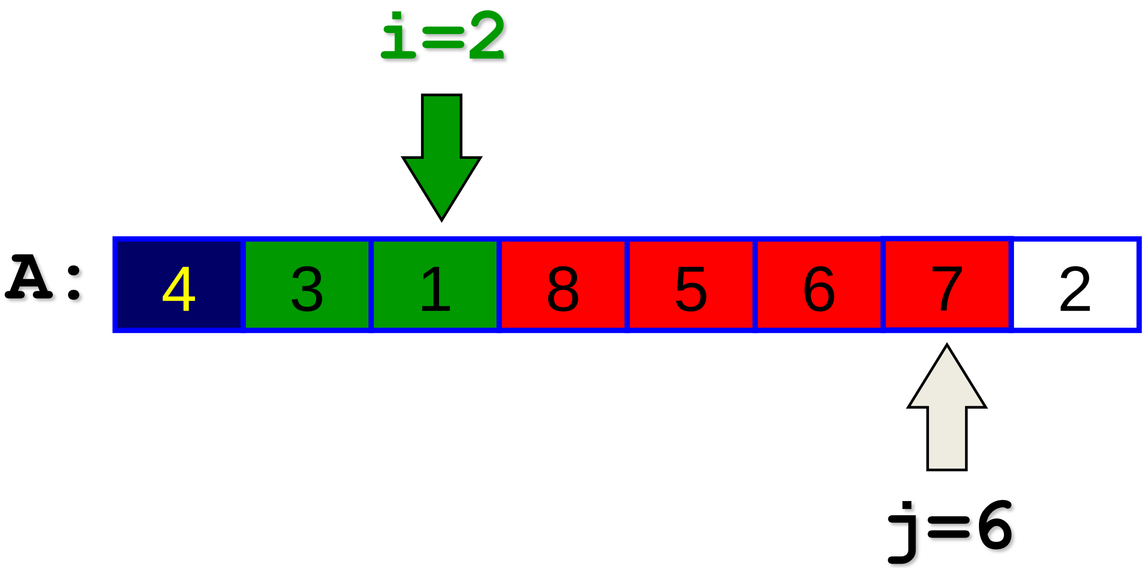
Partition Example



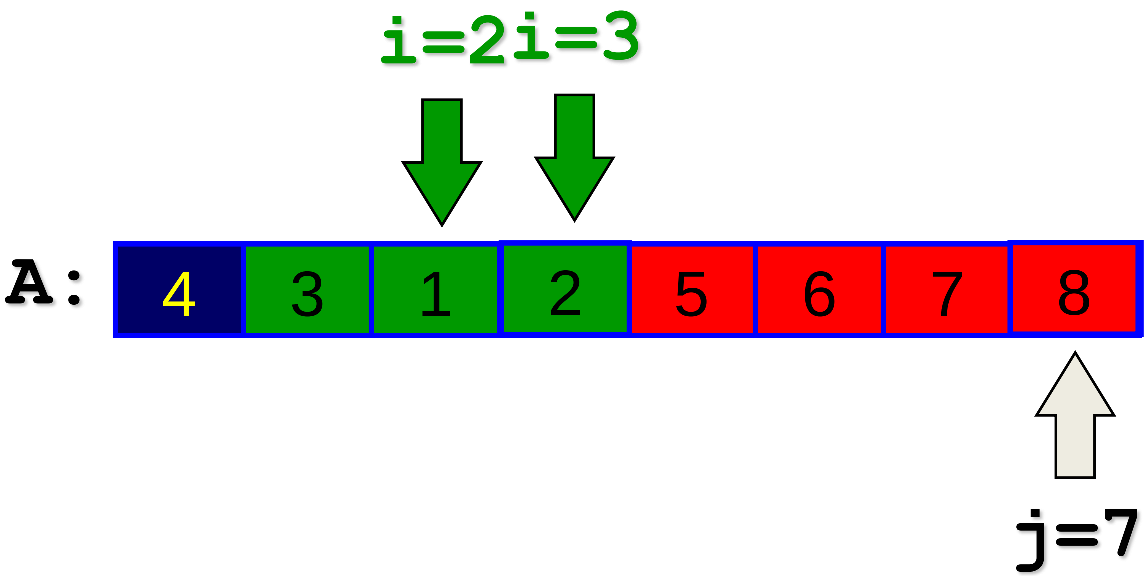
Partition Example



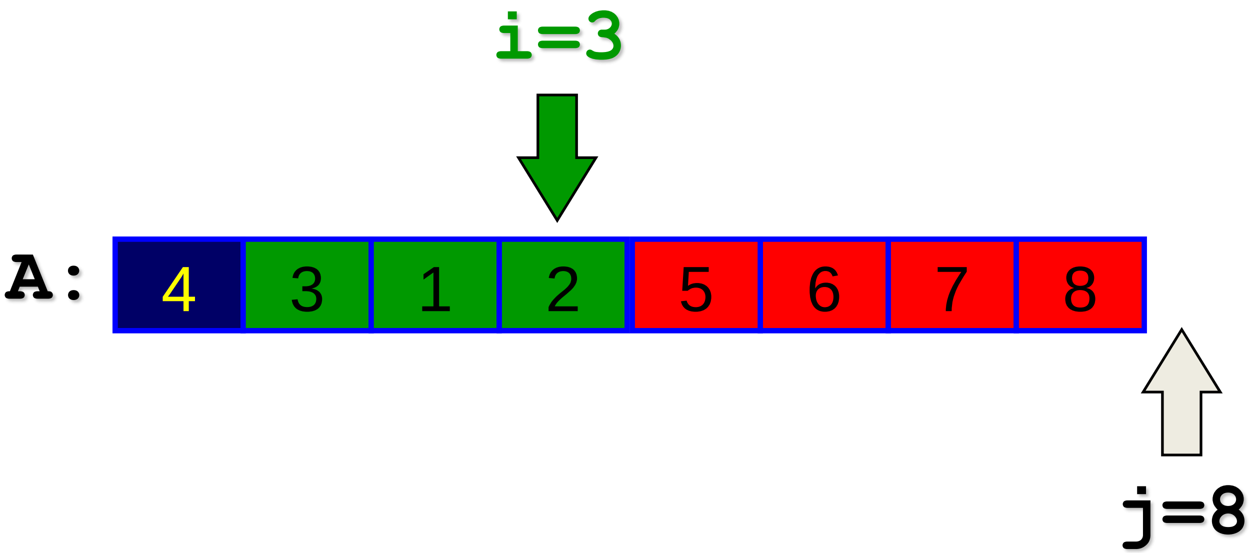
Partition Example



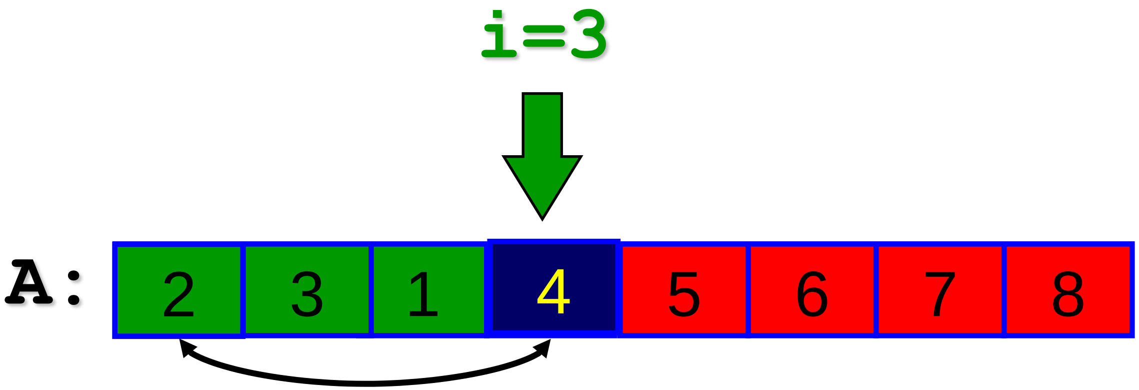
Partition Example



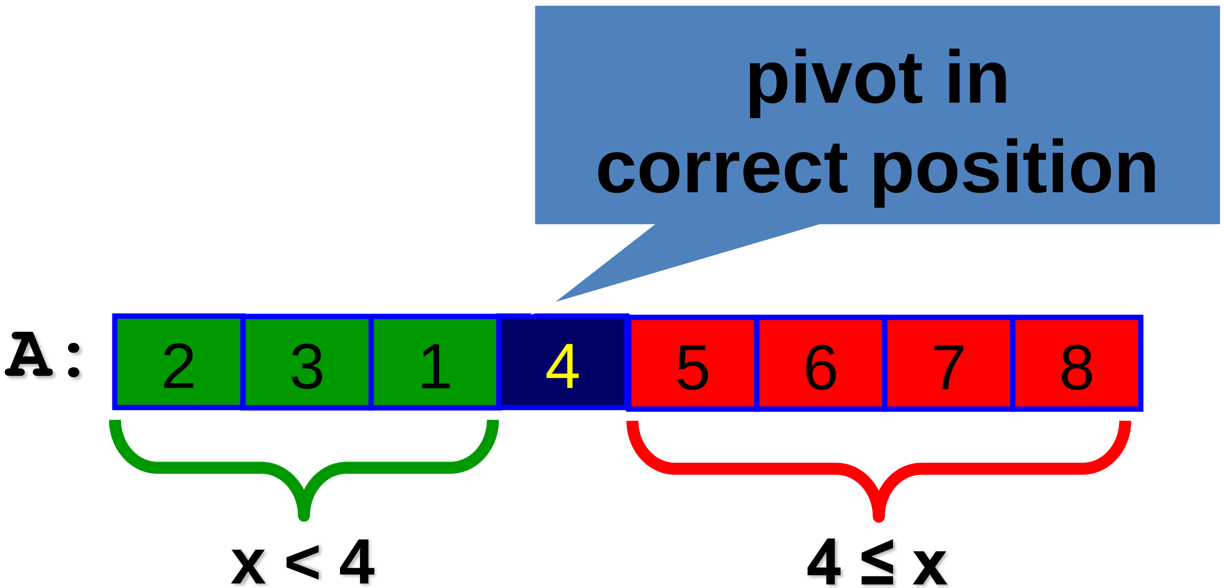
Partition Example



Partition Example

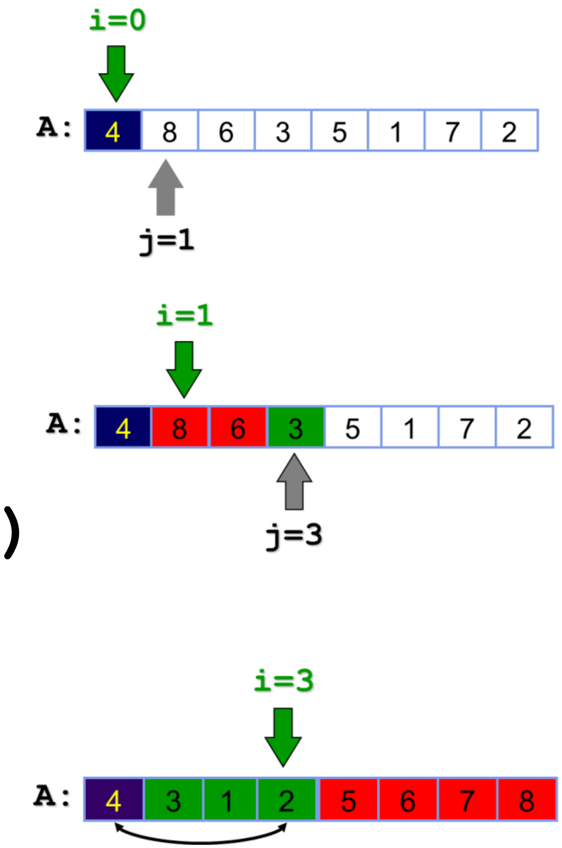


Partition Example



Algorithm of Partition

```
Partition(A, left, right)
    x ← A[left]
    i ← left
    for j ← left+1 to right
        if A[j] < x then
            i ← i + 1
            swap(A[i], A[j])
        end if
    end for j
    swap(A[i], A[left])
    return i
```



n = right – left +1
Time: cn for some constant c
Space: constant

Quick Sort Illustration

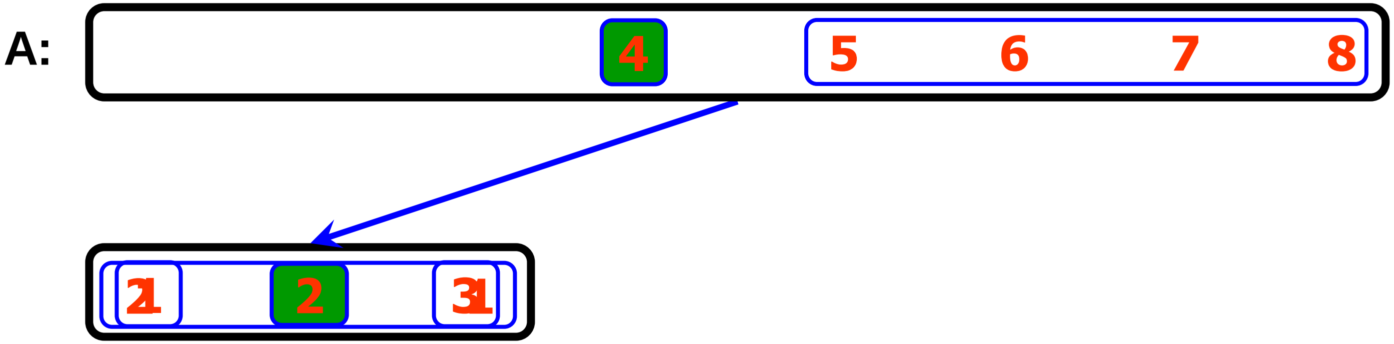
Quick-Sort(A, 0, 7)

Partition

A: 2 8 6 4 55 1 6 77 28

Quick-Sort(A, 0, 7)

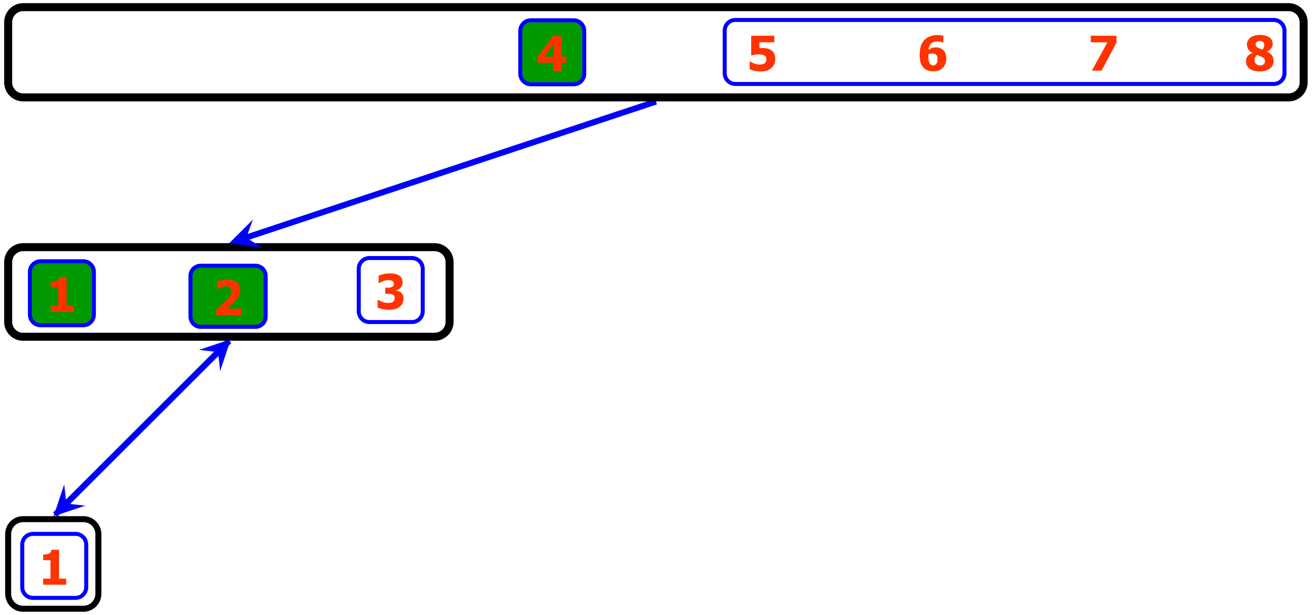
Quick-Sort(A, 0, 2), partition



Quick-Sort(A, 0, 7)

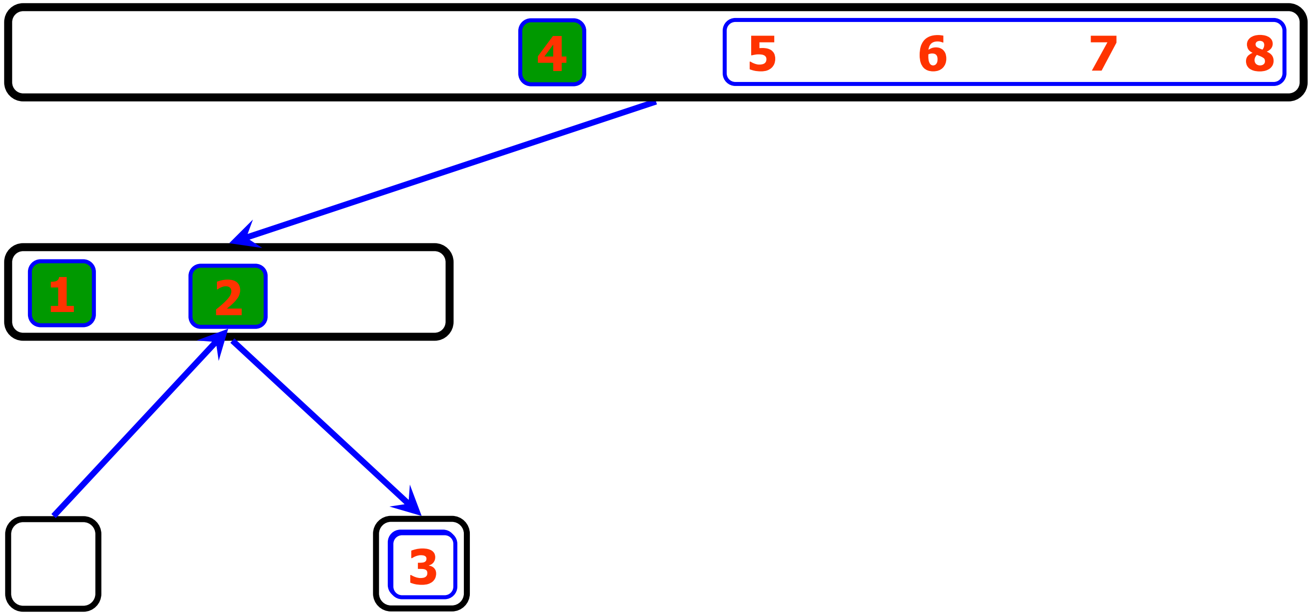
Quick-Sort(A, 0, 0), base case

return



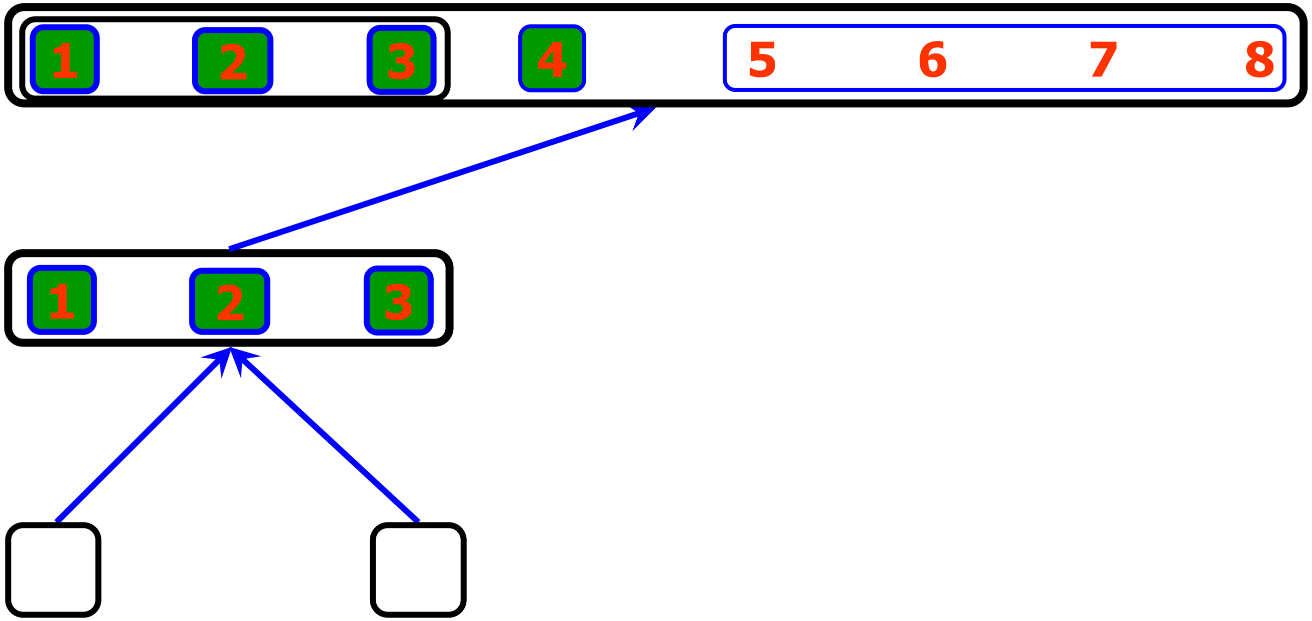
Quick-Sort(A, 0, 7)

Quick-Sort(A, 1, 1) , base case



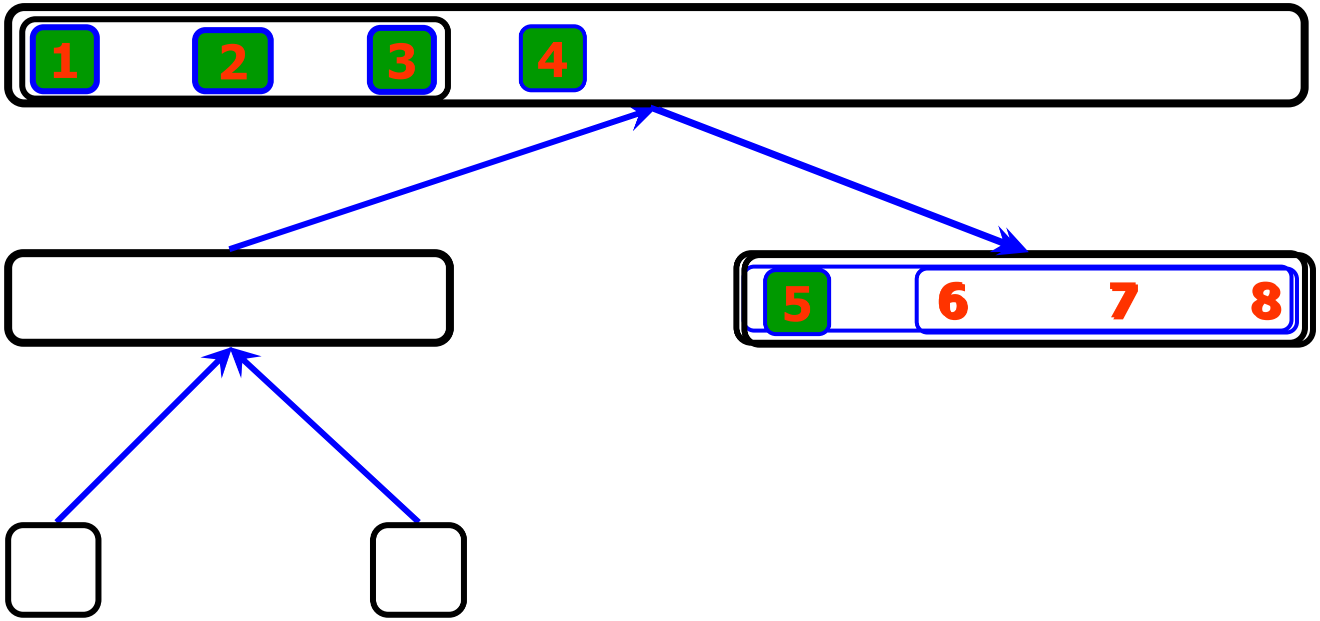
Quick-Sort(A, 0, 7)

Quick-Sort(A, 0, 2), return



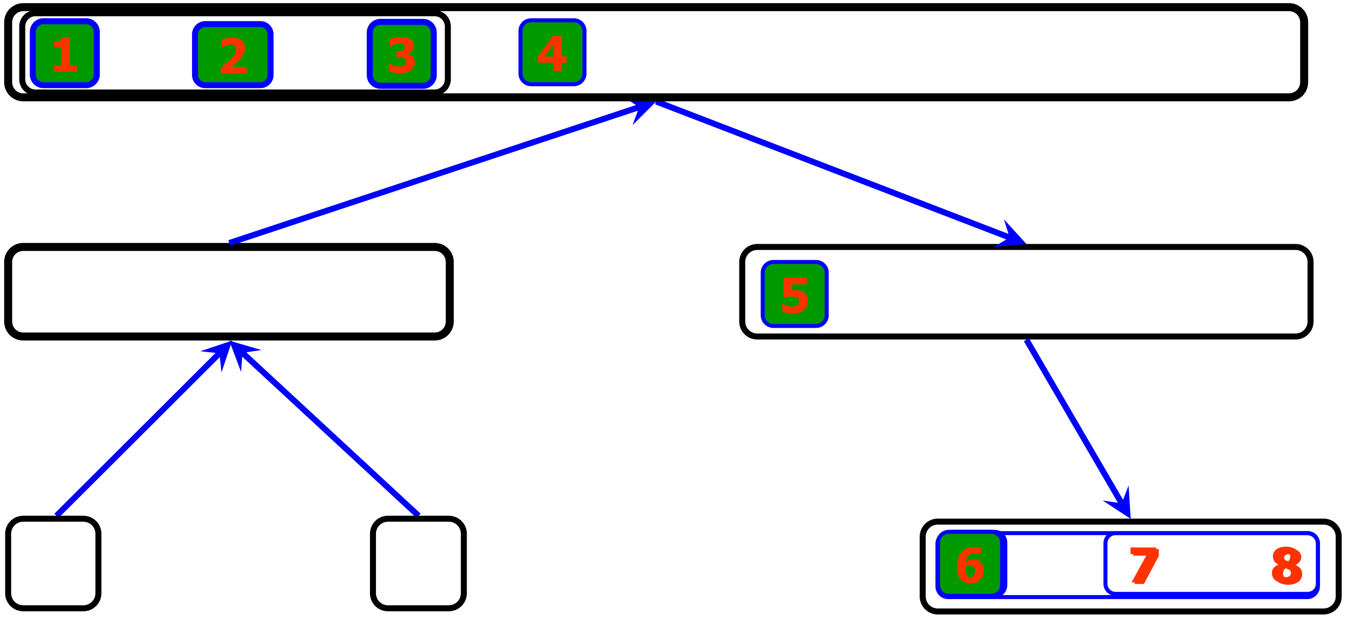
Quick-Sort(A, 0, 7)

Quick-Sort(A, 4, 7) , partition



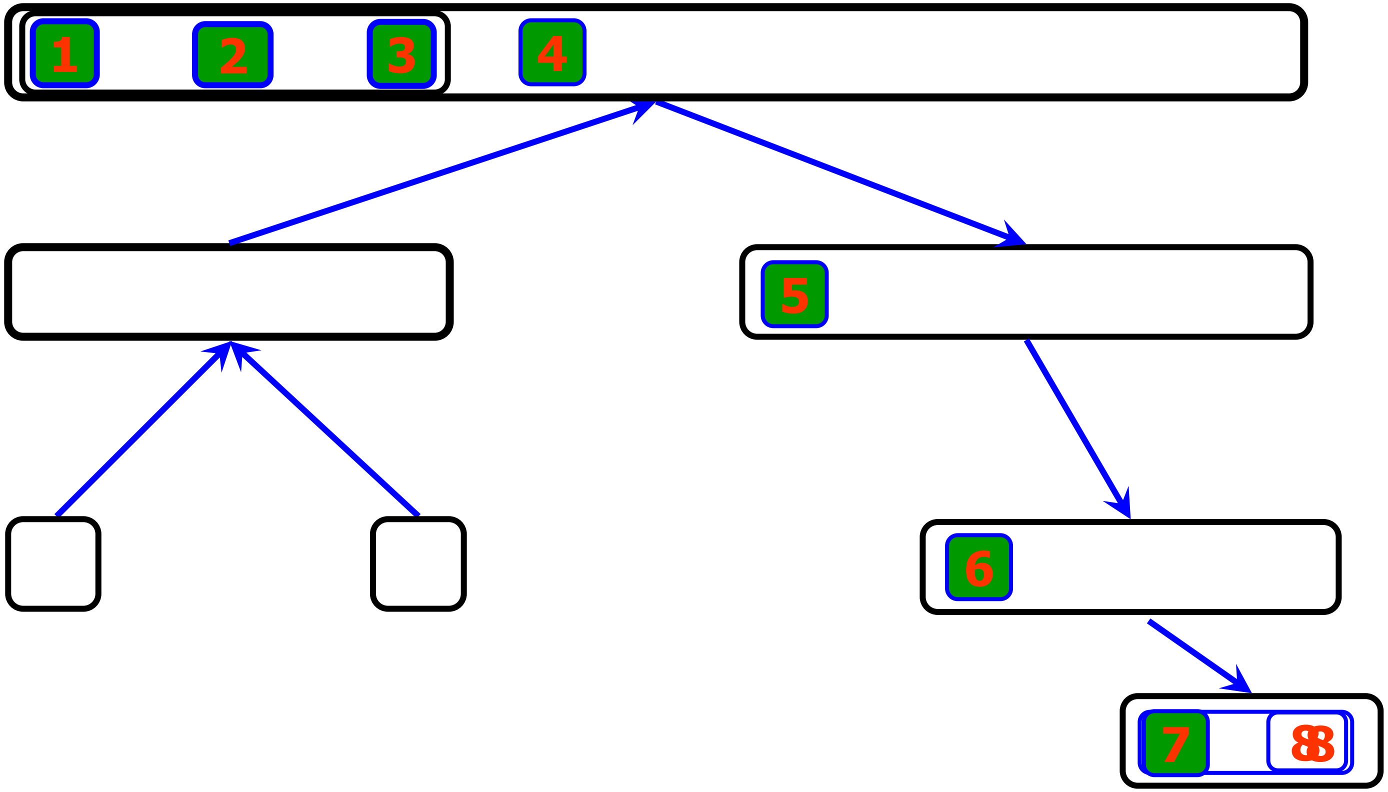
Quick-Sort(A, 0, 7)

Quick-Sort(A, 5, 7) , partition



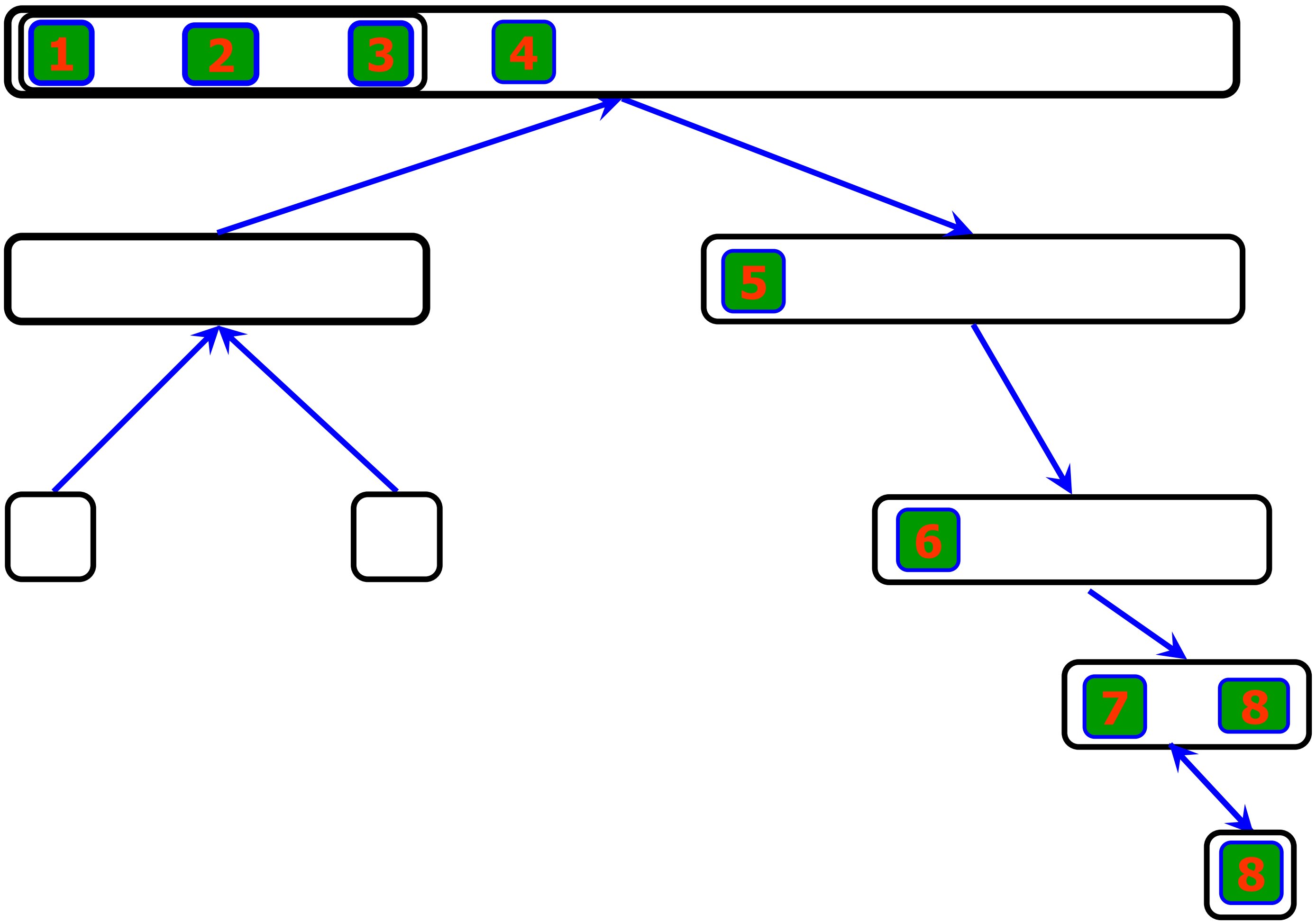
Quick-Sort(A, 0, 7)

Quick-Sort(A, 6, 7) , partition



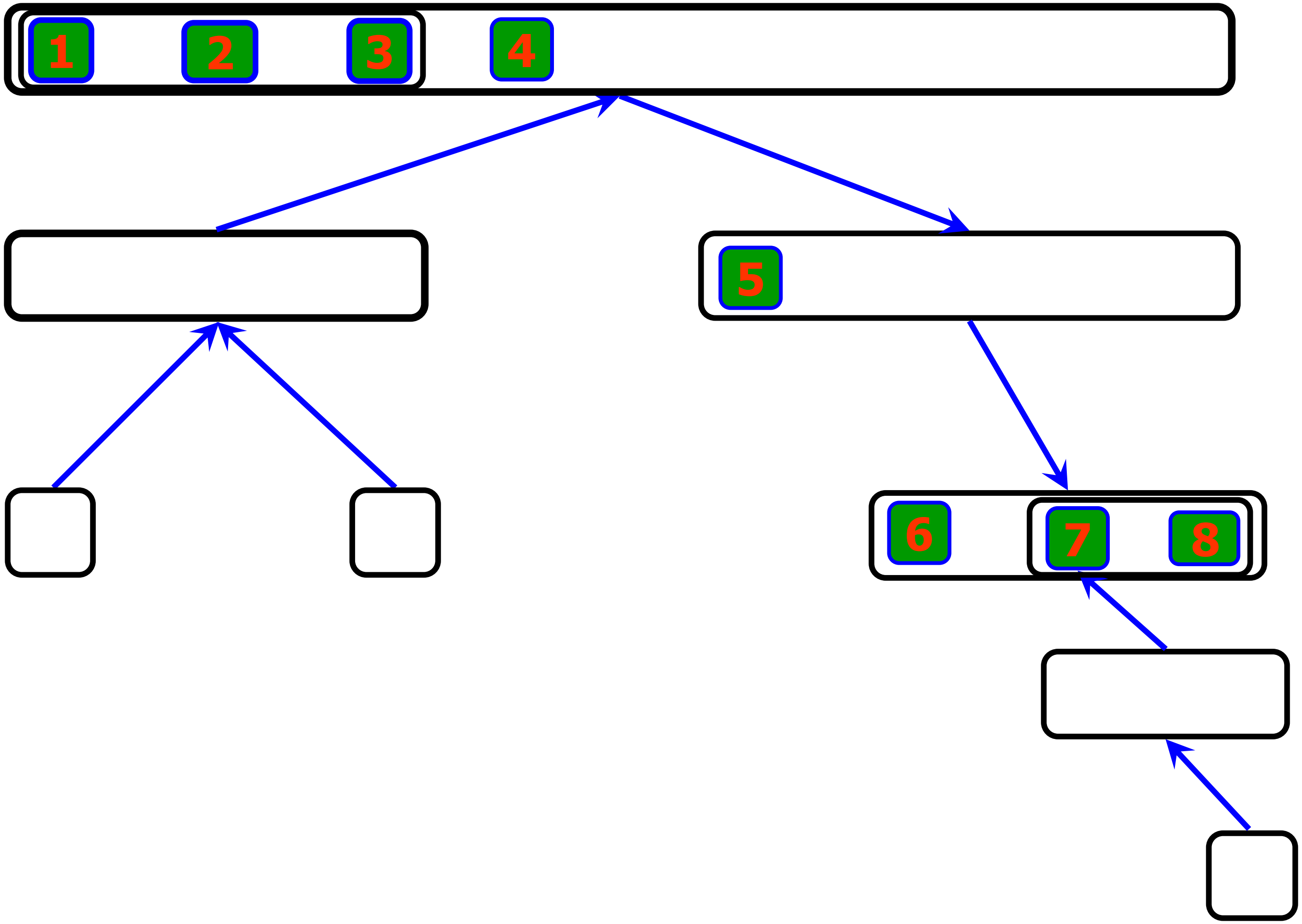
Quick-Sort(A, 0, 7)

Quick-Sort(A, 7, 7) , base case
return



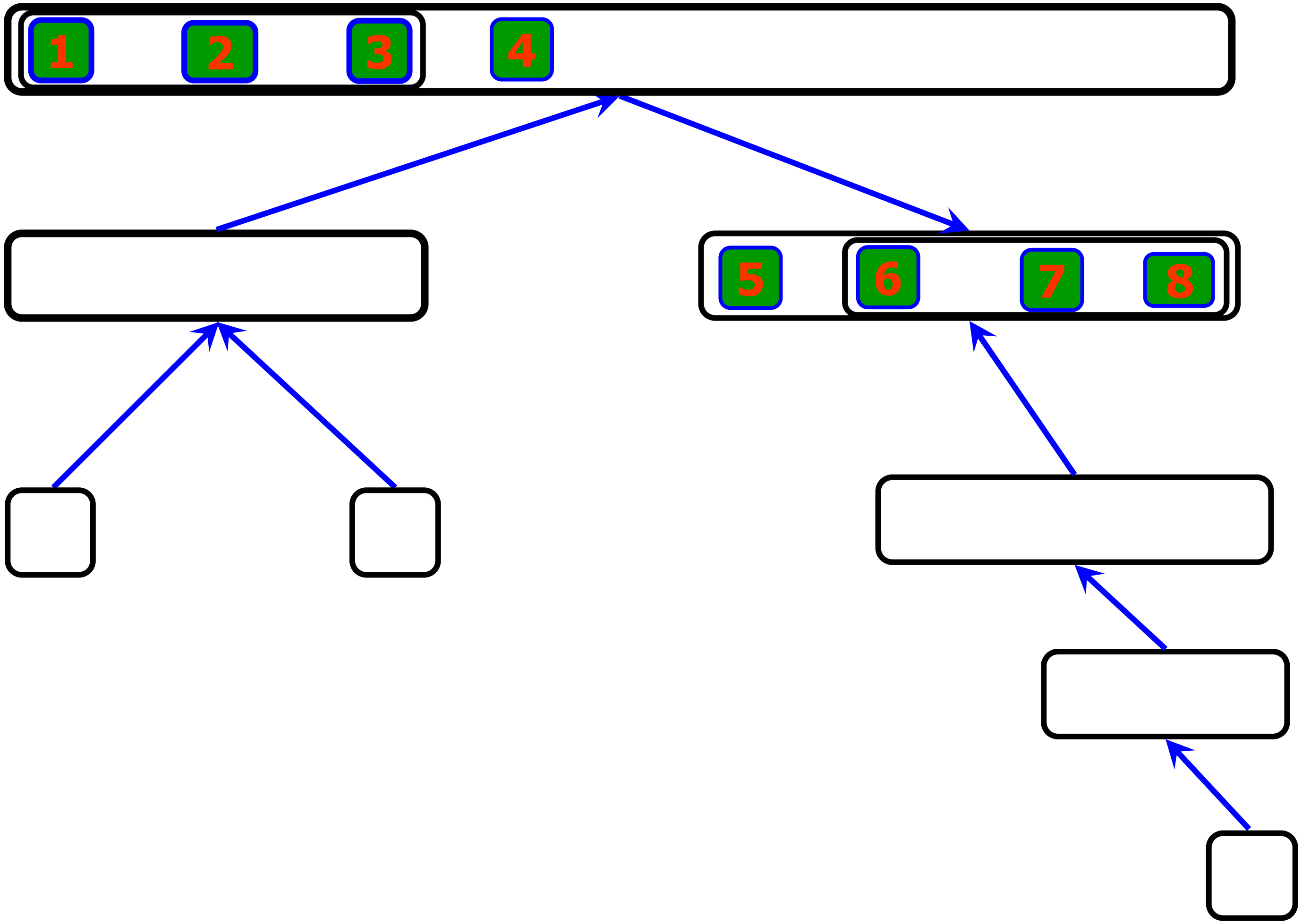
Quick-Sort(A, 0, 7)

Quick-Sort(A, 6, 7) , return



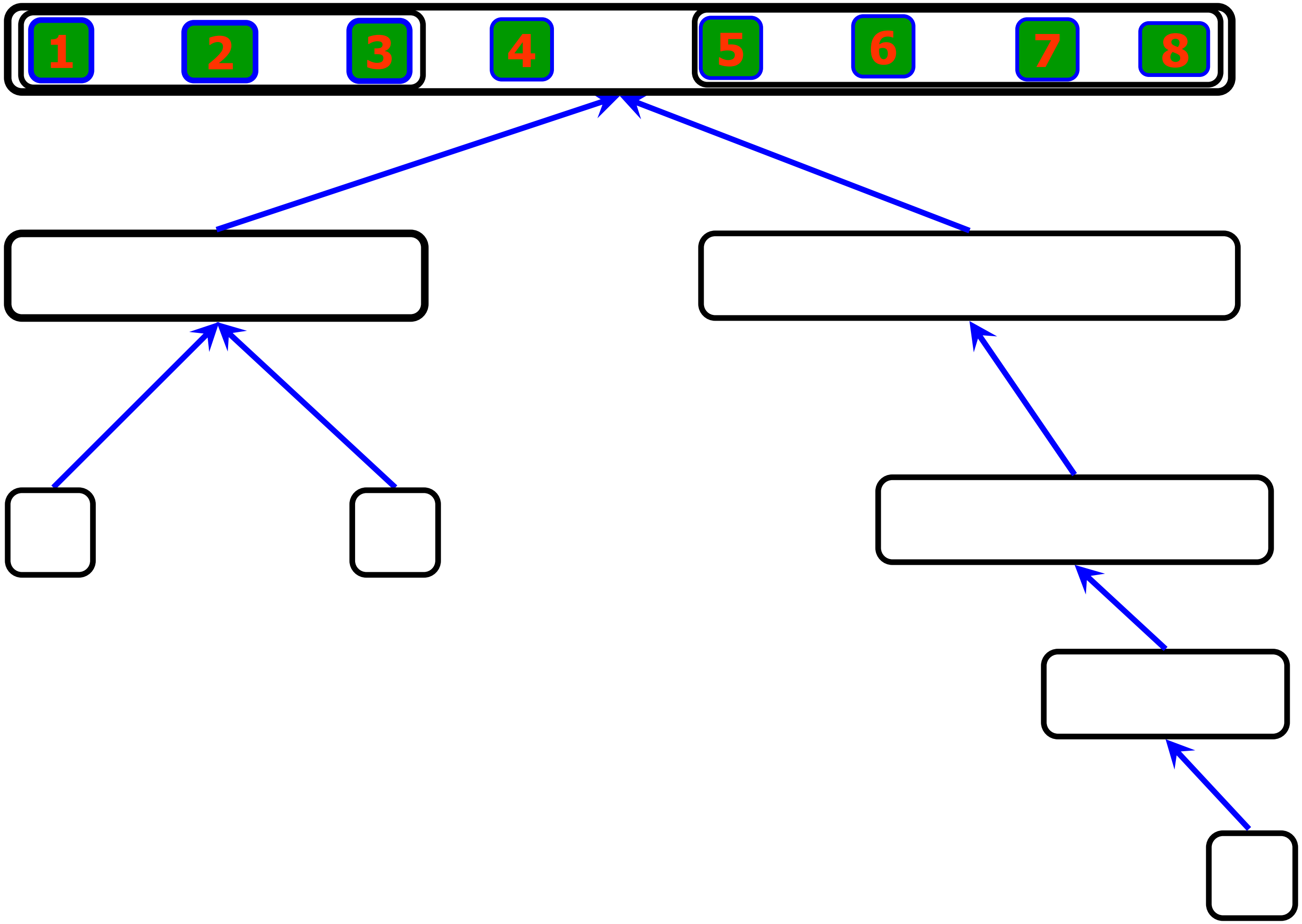
Quick-Sort(A, 0, 7)

Quick-Sort(A, 5, 7) , return



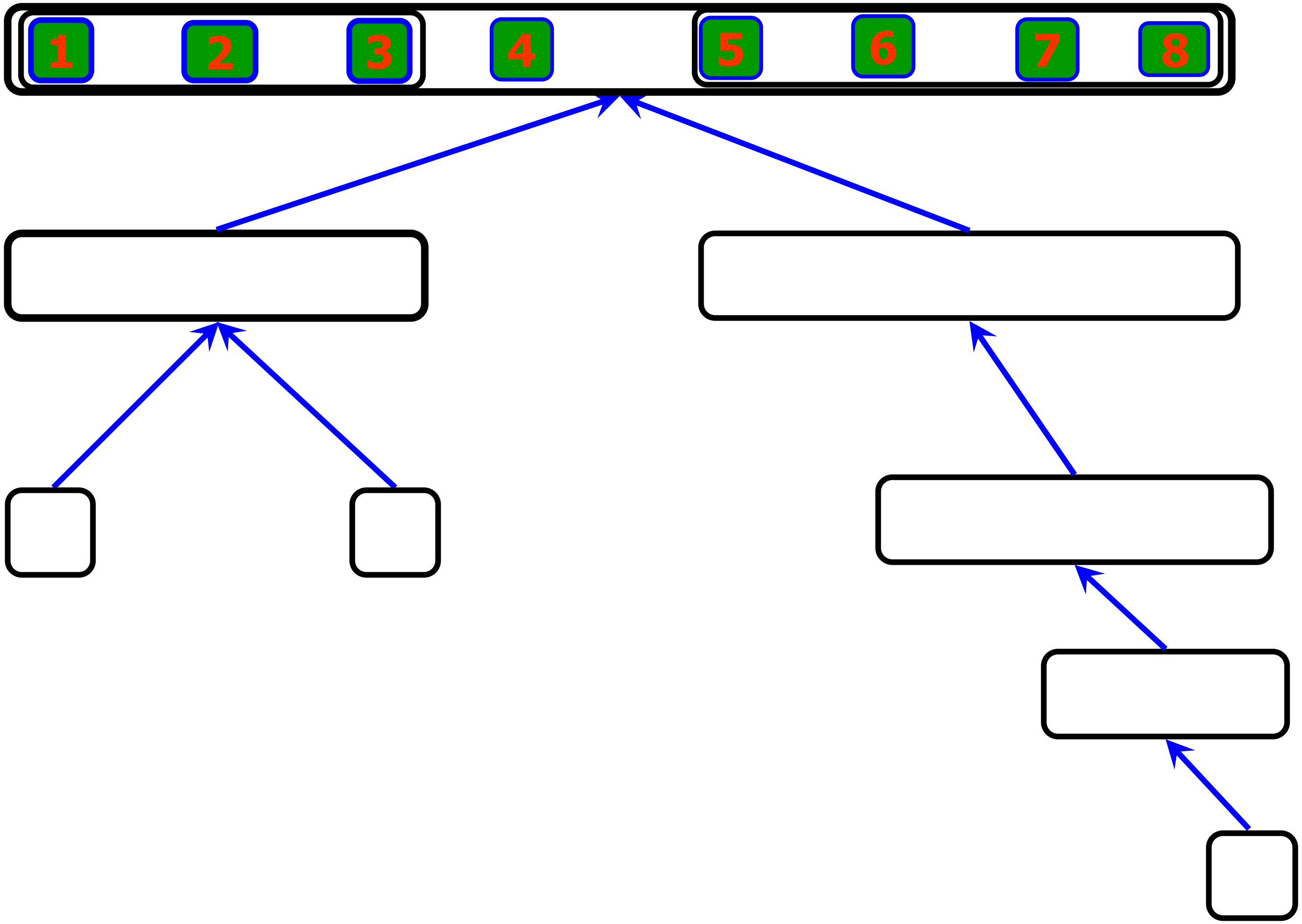
Quick-Sort(A, 0, 7)

Quick-Sort(A, 4, 7) , return



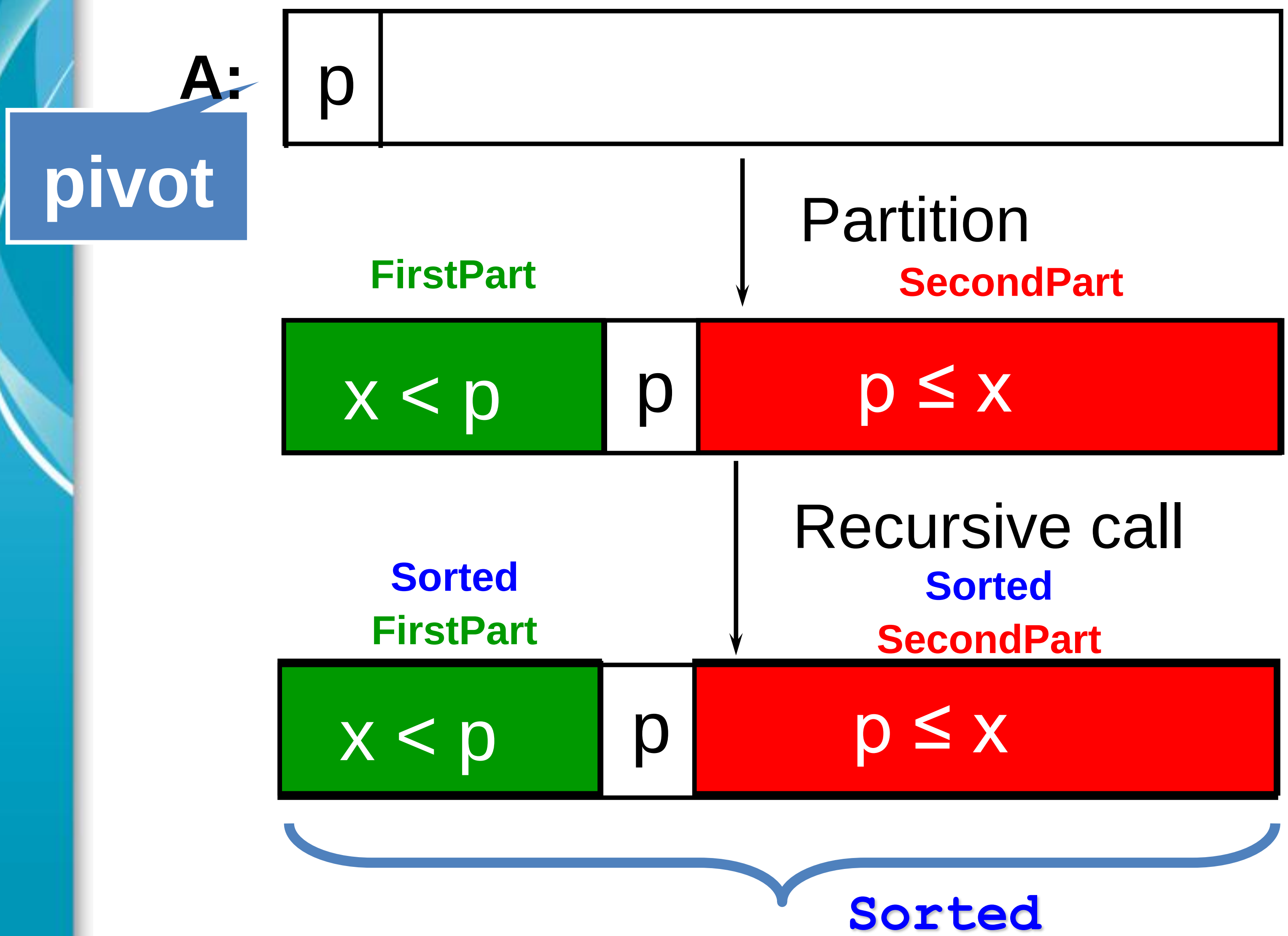
Quick-Sort(A, 0, 7)

Quick-Sort(A, 0, 7) , done!



Quick Sort Algorithm and Coding

Quick Sort


A:
p
pivot
Partition
FirstPart
SecondPart
x < p
p
p ≤ x
Recursive call
Sorted
FirstPart
Sorted
SecondPart
x < p
p
p ≤ x
Sorted

Quick Sort

```
Quick-Sort(A, left, right)
  if    left ≥ right  return
  else
        middle ← Partition(A, left, right)
        Quick-Sort(A, left, middle-1 )
        Quick-Sort(A, middle+1, right)
  end if
```

quickSort

```
# The main function that implements QuickSort
# arr[] --> Array to be sorted,
# low --> Starting index,
# high --> Ending index

# Function to do Quick sort
def quickSort(arr, low, high):
         if len(arr) == 1:
                  return arr
         if low < high:

                  # pi is partitioning index, arr[p] is now
                  # at right place
                  pi = partition(arr, low, high)

                  # Separately sort elements before
                  # partition and after partition
                  quickSort(arr, low, pi-1)
                  quickSort(arr, pi+1, high)
```

Analysis on Quick Sort

- This algorithm is quite efficient for large-sized data sets as its average and worst-case complexity are O(n Log n) and $O(n^2)$.
- Although the worst case time complexity of QuickSort is $O(n^2)$ which is more than many other sorting algorithms like Merge Sort and Heap Sort.
  - QuickSort is faster in practice, because its inner loop can be efficiently implemented on most architectures, and in most real-world data.
- However, merge sort is generally considered better when data is huge and stored in external storage.

Summary of key terms

- Dictionary
- divide and conquer
- Merge sort
  - Merge
- Quick sort
  - Partition